VISUAL BASIC, sodobno programiranje

UROŠ MESOJEDEC

Priročnik za obvladovanje
programiranja z visual basicom

CIP – Kataložni zapis o publikaciji
Narodna in univerzitetna knjižnica, Ljubljana

519.682
519.688

MESOJEDEC, Uroš
Visual basic, sodobno programiranje : priročnik za obvladovanje programiranja z visual basicom / Uroš Mesojedec. - Ljubljana : Pasadena, 1998

ISBN: 961-6065-64-5

75000320

Vsebinsko kazalo

Uvod 1

Komu je knjiga namenjena 1
Razdelitev knjige 1
Kako je knjiga oblikovana 2

Prvi del: Osnove 3

1. poglavje: Namestitev 5

Najrazličnejše različice 5
Boleče neboleča namestitev 6

2. poglavje: Razvojno okolje 7

Zgradba 7
Uvodno okno 7
Osnovna orodja 8
Prilagajanje vmesnika 14
Preurejanje vrstic 14
Preurejanje oken 16
Okno s programsko kodo 16

3. poglavje: Pozdravljen, svet 19

Nastajanje prvega programa 19
Risanje vmesnika 19
Nastavljanje lastnosti 22
Pisanje kode 24
Preizkus 26
Shranjevanje 27
Dopolnitev zgleda 28
Urejanje več gradnikov 29
Dodajanje kode 31
Pokukajmo naprej 34

4. poglavje: Osnove razvoja 37
Okna na hitro 37
Dogodkovno programiranje 38
Gradnik, lastnost, metoda in dogodek 38
Gradnik 38
Lastnost 39
Dogodek in odzivni podprogram 39
Metoda 39

5. poglavje: Uporaba gradnikov 41
Obrazec 41
Lastnosti 41
Metode 45
Dogodki 46
Oznaka 49
Lastnosti 49
Metode 51
Dogodki 52
Vnosno polje 53
Lastnosti 53
Metode 54
Dogodki 54
Gumbi in stikala z okviri 55
Lastnosti 56
Metode 62
Dogodki 62
Seznami 63
Lastnosti 63
Metode 68
Dogodki 69
Drsnika 70
Lastnosti 70
Dogodki 70
Ura 72
Lastnosti 72
Dogodki 72
Menujska izbira 73
Načrtovanje menujev 73
Lastnosti 75
Dogodki 76
Metode 76
Drugi gradniki 77

6. poglavje: Programska koda 79
Elementi kode 79
Simbolična imena 79
Konstantne vrednosti 81
Ločila 83
Komentarji 83
Izrazi in operatorji 84
Prireditveni operator 84
Računski operatorji 84
Primerjalni operatorji 85
Logični operatorji 86
Združevanje nizov 87
Prioriteta operatorjev 87
Podatki in vrste 88
Napoved 89
Doseg in življenjska doba 91
Tipi 95
Sestavljeni tipi 103
Polje 103
Struktura 107
Podprogrami 108
Procedure 108
Funkcije 109
Posredovanje parametrov 111
Rekurzija 112
Klicanje podprogramov 113
Posebni parametri 115
Sklicevanja 117
Nadzor izvajanja 118
Odločitveni stavki 119
Zanke 121

7. poglavje: Predmeti in zbirke 125
Razredi in izvodi 125
Predmetna urejenost 126
Zbirke 126
Vsebniki 128
Predmetna spremenljivka 128
Ustvarjanje in uničevanje predmetov 130
Dinamično ustvarjeni gradniki 131
Uničenje predmeta 133
Predmetni izrazi 133

Drugi del: Delavnica 135

8. poglavje: Okenski komandant 137

Seznami pogonov, map in datotek 138
Lastnosti 138
Dogodki 142
Uporabniški vmesnik 143
Programska koda 145
Sprehod po mapi 145
Nadziranje nizov 148
Uporaba miške in tipkovnice 151
Shranjevanje nastavitev 158
Iskanje napake 159
Upravljanje datotek 163
Uporaba več obrazcev 170
Za konec 178
Vaje 179

9. poglavje: Urejevalnik 181

Pravila poimenovanja 181
Uporabniški vmesnik 183
Dodajanje gradnikov ActiveX 186
Načrtovanje menujev 188
Predloge 189
Standardna pogovorna okna 191
Lastnosti 192
Metode 194
Programska koda 194
Glavno okno 195
Podrejeno okno 196
Urejanje podrejenih oken 199
Uporaba odložišča 200
Preiskovanje besedila 201
Delo z datotekami 204
Prestrezanje napak 212
Prevajanje in podpis 217
Prevajanje 217
Podpis 219
Predmet App 222
Lastnosti 222
Metodi 224
Uporaba funkcij Windows API 224
Branje ukazne vrstice 229
Čarovnije 232
Vaje 237

10. poglavje: Risar 239
Uporaba grafike 240
Koordinatni sistemi 240
Barve 245
Grafični gradniki 247
Lastnosti 247
Metode 250
Dogodki 250
Grafične lastnosti in metode 250
Kje izrisovati? 251
Lastnosti 251
Metode 251
Osveževanje grafike 258
Predmeta Screen in Printer 262
Zaslon 262
Tiskalniki 263
Vmesnik za Okna 95 267
Drsnik in napredek 268
Gor in dol po animacijah 269
Seznam slik 271
Večlična okna, drevesa in seznami 273
Oblikovano besedilo 276
Zgled Risar 278
Uporabniški vmesnik 278
Glavni menu 280
Vzpostavitve ob zagonu 281
Datoteke sredstev 287
Vsebina 287
Orodja 288
Izdelava datoteke sredstev 289
Dodajanje datoteke sredstev projektu 290
Uporaba sredstev 290
Programska koda 292
Ponazoritev stanja 296
Izbira orodja 297
Izbira barve in debeline 298
Risanje likov 299
Obnavljanje 304
Križišča 306
Delo z datotekami 310
Razveljavitev in uveljavitev 316
Prenos na odložišče 317
Izvoz bitne slike 321
Priveski 322

Sprotna pomoč ... 327
Razvoj datoteke s pomočjo ... 327
Uporaba običajne pomoči ... 333
Razpečevanje ... 338
Čarovnik ... 339
Orodja ... 346
Vaje ... 348

Tretji del: Podatki ... 349

11. poglavje: Uvod v zbirke ... 351
Dostop do podatkov ... 351
Zgradba zbirk podatkov ... 352
Gradniki za dostop do podatkov ... 353
Brskanje brez kode ... 354
Lastnosti ... 356
Metode ... 361
Dogodki ... 361
Uporaba predmeta Recordset ... 364
Analiza položaja ... 364
Premiki ... 367
Iskanja ... 368
Dostop do vsebine ... 371
Popravljanje in dodajanje vsebine ... 372
Odstranjevanje zapisov ... 377
Zmogljivi vezani gradniki ... 377
Vstavljeni predmet ... 378
Oblikovani vnos ... 385
Vezana seznama ... 386
Vezani preglednici ... 390

12. poglavje: Predmetni model ... 397
Zgradba ... 397
Opis zbirke podatkov ... 399
Načrtovanje relacijske zbirke ... 399
Izgradnja ... 402
Urejanje ... 408
Vzdrževanje zbirke ... 411
Zgoščevanje ... 412
Popravljanje ... 412
Vsebina ... 413
Razred Recordset ... 413
Podatkovni tipi v zbirki ... 418

Zveza z gradniki ... 420
Transakcije ... 421
Delovni prostori ... 423
Varne seje ... 425

13. poglavje: Jezik SQL ... 433
Raba ... 433
Elementi jezika ... 434
Ukazi ... 435
Določila ... 435
Operatorji in izrazi ... 436
Združevalne funkcije ... 437
Preizkus ... 437
Struktura ... 441
Tabele ... 441
Celovitost podatkov ... 443
Podatki ... 447
Izbor podatkov ... 448
Akcijske poizvedbe ... 453
Sestavljene poizvedbe ... 455
Značilnosti Jet SQL ... 460
Visual basic in SQL ... 462
Posredovanje ukazov ... 462
Pripravljene poizvedbe ... 463
Izgradnja poizvedb ... 466

14. poglavje: Strežnik in odjemalec ... 467
Način dostopa ... 468
Uporaba podatkovnega stroja Jet ... 469
Neposredni dostop ... 469
Gonilniki ODBC ... 470
Uporaba strežnika ... 470
Vzpostavitev podatkovnega vira ... 470
Do podatkov prek stroja Jet ... 473
Do podatkov neposredno ... 478
Značilnosti ... 486
Povezovanje ... 486
Zmanjšanje potrebnih povezav ... 487
Kazalci ... 487

15. poglavje: Razpečevanje ... 491
Predmeti DAO ... 491
Gonilniki ODBC ... 492

Četrti del: Predmeti 493

16. poglavje: Uvod 495

Razredi in predmeti 495
Izrazoslovje 496
Brskalnik 497

17. poglavje: Lastni razredi 499

Razredni modul 499
Ime 500
Vmesnik 500
Življenjska doba 501
Razredni moduli proti običajnim 506
Skrivanje podrobnosti 507
Lastnosti 508
Metode 514
Nadgradnja razreda 516
Privzeti član 516
Dogodki 518
Večličnost 521
Abstraktni razred 522
Izpeljava lastnosti 524
Različni vmesniki 525
Zadolžitve 526
Povezovanje 528
Predmetni model 529
Zgled modela 530
Zbirke 531
Pomočniki 537
Poimenovanje 539

18. poglavje: Gradniki 541

Strežniki 541
Znotrajprocesni strežnik 542
Zunajprocesni strežnik 553
Napotki 569
Gradniki vmesnika 572
Spoznavanje 573
Izdelovanje gradnikov 580
Napotki 596
Prevajanje 602

19. poglavje: Razpečevanje 605
Vklop strežnika 605
Namestitev zunajprocesnega strežnika 605
Namestitev odjemalca 606
Prenos prek spleta 607

Peti del: Svet 611

20. poglavje: Izpostavljeni razredi 613
Uporaba gradnikov 614
Sklicevanje 615
Uporaba vmesnika 616
Sproščanje 617
Obravnavanje izvajalnih napak 617
Prezaposlenost 619
Zaposleni gradnik 619
Čakajoča zahteva 619
Lastna sporočila 620
Vizualno delo 621

21. poglavje: Narečja 623
VBA 623
Pisarna 624
VBS 624
Elementi jezika 625
Pripravljeni razredi 626
Upravljanje 626
Spletni programi 628

Dodatek A: Priložena plošča 635

Dodatek B: Opis datotek 637

Stvarno kazalo 641

Uvod

Z visual basicom sva že stara znanca. Od začetka najinega znanstva sva skupaj preživela že kar nekaj vzponov in padcev. In ko sem že mislil, da se ga ne bom več dotaknil, me je presenetil z razredi. Je že tako, da je z visual basicom še vedno daleč najbolj enostavno posegati v najvznemirljivejše tehnološke posladke, ki jih za vesoljno občestvo uporabnikov osebnih računalnikov snujejo v Redmondu. Naj bodo tudi vam v slast.

Komu je knjiga namenjena

Morda bi bilo bolje vprašati, komu ni. Visual basic je v rabi vsepovsod, od povsem enostavnih »skriptnih« različic, ki omogočajo, da preimenujemo kako datoteko ali popestrimo spletne strani, do najzmogljivejše različice Visual Basic Enterprise Edition, ki omogoča izgradnjo oddaljenih strežnikov za avtomatizacijo ActiveX, ki lahko črpajo podatke iz zmogljivih podatkovnih strežnikov. Ta knjiga se trudi ustreči vsem.

Posebnega predznanja, razen uporabe okolja Windows in kake angleške besede, ne potrebujete. Vsekakor pa vas bo ta knjiga skušala navdušiti, da bi si dodatnega znanja nabrali čim več. Vsi zgledi v njej so pripravljeni tako, da vas poizkušajo pripraviti do nadaljnjega poizkušanja in ugotavljanja novih in novih zmožnosti orodja, ki ga boste uporabljali.

Razdelitev knjige

Knjiga je razdeljena na pet delov, ki pokrivajo naslednje:

Osnove Spoznavanje osnov visual basica.

Delavnica Razvoj treh zaokroženih zgledov, ki ponazarjajo temeljne programerske tehnike.

Podatki Tretji del je v celoti posvečen razvoju programov, ki izkoriščajo zunanje vire podatkov (*databases*).

Predmeti V četrtem delu se bomo posvetili morda najbolj vznemirljivi plati visual basica – razvoju lastnih razredov.

Svet Za konec bomo v petem delu spoznali najrazličnejše rabe narečij basica, od rabe v pisarniških programih do izdelovanja dejavnih spletnih strani.

Zaradi obsega obravnavane tematike se razlage ne ponavljajo iz poglavja v poglavje, in je knjigo priporočljivo vsaj enkrat prebrati od začetka do konca. To še posebej velja za bralce, ki se z visual basicom šele spoznavajo.

Poznavalci visual basica in tisti, ki bodo knjigo vsaj enkrat prebrali v celoti ter preizkusili priložene zglede, pri uporabi posameznih poglavij iz kateregakoli dela knjige najverjetneje ne bodo imeli posebnih težav.

Kako je knjiga oblikovana

V knjigi je uporabljenih več različnih pisav in nekaj posebnih simbolov, ki dodatno ponazarjajo obravnavano tematiko. Značilni zgledi so našteti spodaj:

- Posebej pomembni izrazi so zapisani *ležeče*. Manj pogosti slovenski prevodi angleških izrazov so razloženi tudi z izvirnikom (*original*). Pomemben člen jezika je pred razlago izpisan **polkrepko**.
- Izbiranje menujev, jezičkov, stikal, gumbov ... je ponazorjeno z zaporedjem:

  Tools ➪ Options ➪ jeziček **Editor**

- Oznake, zgledi in programski izpisi so prikazani v `nesorazmerni pisavi`:

```
MsgBox "Pozdravljen, svet!"
```

- Pomembnejši deli programske kode, ki se nanašajo na spremljajočo razlago, so označeni **`polkrepko`**:

```
MsgBox CStr(p1)
```

- Za opisni prikaz pravil jezika basic rabijo simbolični zapisi. Neobvezni del je označen z oglatimi oklepaji (`[]`):

```
MsgBox strPoziv[, nVidez][, strNaslov]
```

- Če je mogoče izbirati med več možnostmi, jih oklepata zavita oklepaja (`{}`), razmejuje pa navpična črta (`|`). Če v *opisnem* prikazu nastopa rezervirana beseda, je označena **`krepko`**, simbolični del pa je izpisan *`ležeče`*:

```
[{Private|Public}] Const = vrednost
```

NAMIG Namigi postrežejo z dodatnimi informacijami, ki ponavadi presegajo okvir trenutno obravnavane tematike, vendar se vsebinsko navezujejo nanjo.

PRVI DEL

Osnove

V TEM DELU

Namestitev
Razvojno okolje
Pozdravljen, svet
Osnove razvoja
Uporaba gradnikov
Programska koda
Predmeti in zbirke

Pričujoči del knjige pojasnjuje nameščanje in prilagajanje, postreže z opisom vmesnika, predstavi način vizualno-dogodkovnega programiranja ter spopad človeka in stroja s prvim pravim programom v visual basicu. Sledi podrobnejše pojasnjevanje temeljnih načel programiranja v visual basicu in podroben pregled osnovnih gradnikov in programerskih stavkov. Za konec pride na vrsto še ogled osnovnih vgrajenih predmetov in njihove rabe.

Preden začnemo, samo še kratko opozorilo za bolj izkušene uporabnike visual basica. Ta del knjige se bolj ali manj pretvarja, da visual basic ni predmetno razvojno orodje, in se izogiba omenjanju predmetov. Za manj zahtevne uporabnike, predvsem za začetnike, je pomembno, da najprej spoznajo osnove razvijanja programov v visual basicu. Šele ko bodo stvari, obravnavane v prvem delu knjige, jasne in razumljive, se bodo lahko spopadli s predmetnim visual basicom, opisanim v nadaljevanju.

1. POGLAVJE

Namestitev

Visual basic je naprodaj v kar nekaj različicah. Najpreprostejša je celo brezplačno na voljo v svetovnem spletu (`http://www.eu.microsoft.com/vbasic/download/cce/`). Ne glede na to, katero različico imate, izberite popolno namestitev vseh sestavnih delov. Tako si zagotovite kar največjo uporabnost, dodobra izkoristite vložene denarce, res pa je, da se hkrati poslovite tudi od kar največjega dela diska.

Najrazličnejše različice

Oglejmo si malo natančneje, kaj vse nam Microsoft ponuja pod zaščiteno blagovno znamko Visual Basic:

Različica	Cena	Namen
Control Creation Edition	Brezplačno.	Samo razvoj gradnikov.
Learning Edition	Okrog 27.000 SIT.	Učenje visual basica.
Professional Edition	Okrog 130.000 SIT.	Najuporabnejša različica.
Enterprise Edition	Okrog 320.000 SIT.	Najpopolnejša različica, njeno ceno pa je treba dodobra upravičiti.
for Applications	Sestavni del nekaterih namenskih programov.	Nadzor delovanja namenskih programov.
Scripting Edition	Sestavni del nekaterih sistemskih orodij.	Nadzor delovanja sistemskih orodij.

V tej knjigi si bomo ogledali osnove, ki so skupne vsem različicam, kar precejšen del pa je je posvečen tudi zahtevnejšim zmožnostim visual basica, ki jih najdete le v različicah Professional in Enterprise. Vsekakor za resno delo potrebujete vsaj različico Professional.

Boleče neboleča namestitev

Če niste prebrali uvodnega odstavka, naj ponovim: najsi imate to ali ono različico, namestite jo v celoti (če vas namestitveni program kaj sprašuje, izberite *Complete Installation*). Nekatere različice visual basica se celo ne bodo uspele pravilno namestiti, če boste preveč domišljavo izklapljali ali prestavljali sestavne dele paketa. Zmogljivejše različice pa prinašajo še eno hudo težavo: zahtevajo silno veliko prostora natančno na tistem disku, na katerem so nameščena tudi Okna (recimo mu osnovni disk). Čeprav za to ni nobenega razloga, mora postopek nameščanja prenesti deljene knjižnice prav v sistemsko mapo Oken. Očitno bi Bill rad, da bi osnovni disk poimenovali kar Microsoft in bi vanj ne nameščali programske opreme nobenega drugega izdelovalca. Skratka, namestitev lahko spodleti, če imate na osnovnem disku premalo prostora, pa čeprav je preostalo diskovje popolnoma prazno.

Ko se vam visual basic namesti, lahko s plošče CD-ROM, priložene knjigi, namestite še popravke *Service Pack*, ki popravijo napake iz nekaterih gradnikov in izboljšajo prevajalnik najboljše različice visual basica.

2. POGLAVJE

Razvojno okolje

Visual basic je že od začetkov združeno razvojno okolje (IDE, *Integrated Development Environment*), ker združuje večino funkcij razvijanja programov, kot so npr. urejanje grafičnega vmesnika, urejanje kode, prevajanje, poskusno izvajanje (razhroščevanje) itn. Uporabniški vmesnik se v novih različicah ni bistveno spreminjal, posnemala pa so ga skoraj vsa konkurenčna orodja, kar je nedvomno znak, da je zelo premišljen in funkcionalen. Pa vendar je čisto vse mogoče izboljšati. V visual basicu smo tako kar nekaj časa pogrešali drobne malenkosti, s katerimi so se dičili konkurenčni izdelki. V peti različici so razvojno okolje temeljito prenovili. Visual basic nam skuša pomagati na vsakem koraku in je v mnogih podrobnostih spet za korak pred tekmeci. No, nekaj malenkosti je še treba izpiliti, a morda so jih celo nalašč zanemarili, da imajo dober izgovor za novo različico.

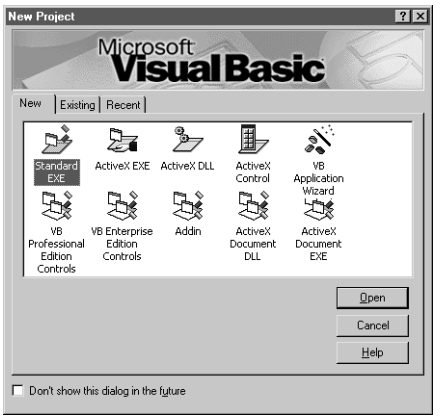
Zgradba

V nadaljevanju bomo spoznali ključne elemente razvojnega okolja. Posebej si bomo ogledali vsako izmed pomembnejših orodij, ki so nam na voljo med delom, in spoznali, kako visual basic prilagodimo svojim željam in potrebam, da nam bo v kar največjo pomoč.

Uvodno okno

Po uspešni namestitvi boste ikono za zagon visual basica našli v menuju **Start**, skupaj z drugimi programi (**Programi**/**Programs**). Ob prvem zagonu nas pozdravi uvodno okno (slika 2-1).

Kako pestre so sličice, je odvisno od tega, katero različico visual basica uporabljate (glej *Najrazličnejše različice* v prejšnjem poglavju). Slika 2-1 je iz najboljše različice in se morda razlikuje od tiste na vašem zaslonu. Uvodno okno omogoča, da se lotimo novega projekta ali se naprej ukvarjamo s starim projektom, ki smo ga bili uspešno shranili. Sestavljajo ga tri strani, eno od njih pa izberemo z jezičkom na vrhu. Najprej imamo pod jezičkom **New** na voljo, da se lotimo novega projekta. Vsaka posamezna izpeljanka projekta je predstavljena z lastno sličico. Jeziček **Existing** nam omogoča, da odpremo že shranjeni projekt, jeziček **Recent** pa bi moral biti zaenkrat še prazen. Tam se zbirajo *shranjeni* projekti, s katerimi smo se nazadnje



Slika 2-1: Uvodno okno omogoča, da se lotimo novega projekta ali nadaljujemo prejšnjega.

ukvarjali. S stikalom spodaj (**Don't show this dialog in the future**) lahko izklopimo prikazovanje uvodnega okna. Vse funkcije uvodnega okna so dostopne tudi prek dveh menujskih ukazov. Kljub temu je uvodno okno koristno in vam ne priporočam, da bi ga izklopili.

Če je vse nastavljeno tako, kakor je prikazano na sliki zgoraj (jeziček New, izbrana možnost Standard EXE), kliknite gumb **Open**.

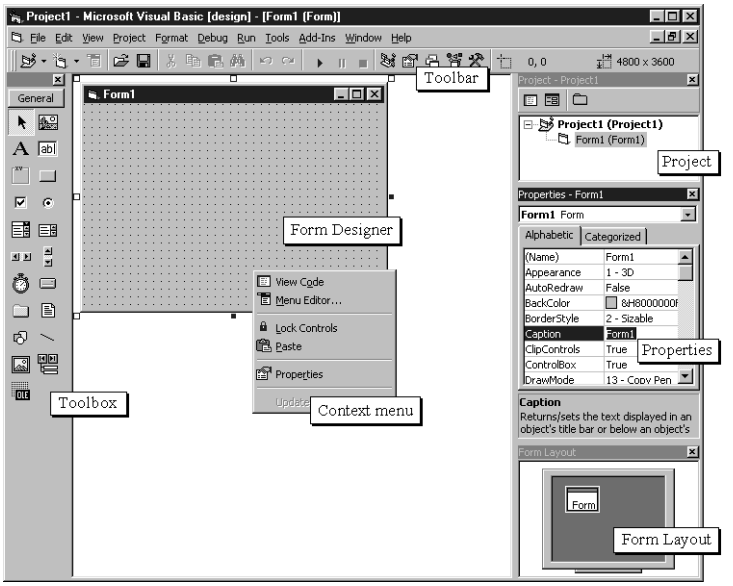
Osnovna orodja

Po tej uvodni odločitvi nas pričaka popolno, integrirano razvojno okolje, v katerem je na voljo kar nekaj pomožnih orodij. V peti različici visual basic posnema videz razvojnega okolja *Developer Studio*. Najbolj očitna podobnost med njima je ves čas vidno glavno okno, v katerem so zbrana vsa druga. Čeprav lahko še vedno preklopimo tudi v staro obliko, ponuja nova toliko prednosti, da se vam bo nedvomno kmalu prikupila. Na sliki 2-2 so označeni osnovni elementi vmesnika.

NAMIG Nova oblika vmesnika omogoča, da se hkrati ukvarjamo z več projekti (MDI, *Multiple Document Interface*), podobno kakor lahko npr. v Wordu urejamo več dokumentov. Mogoče je preklopiti tudi v staro obliko vmesnika, v kateri se je mogoče ukvarjati le z enim projektom, vendar to odsvetujem. Kdor to vseeno želi, naj izbere naslednje ukaze glavnega menuja: **Tools** ➪ **Options** ➪ jeziček **Advanced** ➪ stikalo **SDI Development Environment**. Vmesnik SDI (*Single Document Interface*) je nekoliko neroden, ker nima določenega ozadja in lahko po pomoti, zgolj če napačno kliknemo z miško, preklopimo v kak program iz ozadja. Za povrh lahko kako okno tudi izgubimo z zaslona, če je projekt ustvaril kdo z večjo zaslonsko ločljivostjo.

Še stavek ali dva o najpomembnejših elementih.

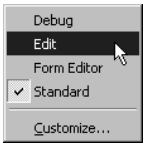
Slika 2-2: Elementi združenega razvojnega okolja (IDE) visual basica.



Priročni menu

Priročni menu (*Context menu*) prikličemo, če kliknemo desno tipko miške. Njegova vsebina se spreminja glede na predmet oz. območje, nad katerim smo kliknili z miško. Na sliki je prikazan priročni menu okna Form Designer. Ukazi v njem so dosegljivi tudi v glavnem menuju programa, vendar so tu priročno zbrani, zato bomo v nadaljevanju priročne menuje pogosto uporabljali.

Slika 2-3: Priročni menu vrstice gumbov (*Toolbar*)



Vrstica z gumbi

V vrstici z gumbi (*Toolbar*) se skrivajo bližnjice do najuporabnejših menujskih ukazov. Ker visual basic nima posebne vrstice stanja, je ta vključena deloma v naslovno vrstico okna, deloma pa v vrstico z gumbi. Tako si lahko v naslovni vrstici preberemo, s katerim obrazcem katerega projekta se trenutno ukvarjamo, še posebej pomembna pa je besedica v oglatem oklepaju, ki ponazarja eno izmed treh stanj razvojnega okolja. Ta stanja so lahko:

- **[design]** – načrtovanje programa in pisanje kode, t. i. *oblikovanje*;
- **[run]** – izvajanje programa, t. i. *izvajanje*;
- **[break]** – začasna prekinitev izvajanja, t. i. *razhroščevanje*.

Skrajno desno v vrstici z gumbi je še merilec položaja in velikosti trenutno risanega gradnika, če pa smo v urejevalniku kode, se na tem mestu izpisujeta vrstica in stolpec, v katerem je kazalec vnosne točke.

Poleg vrstice z gumbi, ki jo vidite na sliki, pozna visual basic še tri orodjarne. Naštete so v pripadajočem priročnem menuju (slika 2-3), v katerem jih lahko vklapljamo in izklapljamo. Če na zaslonu uporabljate visoko ločljivost (vsaj 1024 × 768 pik), je priporočljivo vklopiti tudi več vrstic z gumbi. Vrstica Standard je splošno uporabna, pomen drugih pa naštevam spodaj:

- Debug – v njej so gumbi ukazov za razhroščevanje. Osnovni gumbi te vrstice so tudi v vrstici Standard;
- Edit – gumbi ukazov za pomoč pri urejanju programske kode;
- Form Editor – gumbi ukazov za pomoč pri urejanju gradnikov na obrazcu. Nekatere izmed njih uporabljamo kar pogosto, zato priporočam, da to vrstico vklopite.

Podobno kakor v drugih Microsoftovih programih lahko vrstice z gumbi poljubno preurejamo, dodajamo nove in brišemo stare. Preurejanje dosežemo z ukazom iz priročnega menuja, **Customize**. Če imate čas, se poigrajte z njim.

> **NAMIG** Če uporabljate večjo zaslonsko ločljivost, lahko vklopite prikaz večjih sličic. Uporabiti morate že omenjeni ukaz priročnega menuja vrstice z gumbi **Customize** ➪ jeziček **Options** ➪ stikalo **Large icons**. Da bo delo bolj zabavno, se lahko poigrate tudi z nastavitvami s seznama **Menu animations**.

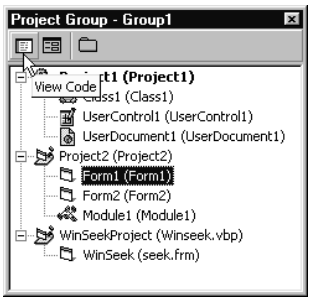
Orodjarna

V orodjarni (*Toolbox*) so *gradniki*, s katerimi oblikujemo uporabniški vmesnik nastajajočega programa. Gradnik (*control*) je že pripravljen sestavni del programa, namenjen točno določenemu opravilu. Večino gradnikov vidimo tudi med izvajanjem programa, nekateri izmed njih (npr. ura) pa nimajo uporabniškega vmesnika, zato so vidni samo med načrtovanjem, ko jih ponazarja le njihova ikona. Izbor gradnikov je odvisen od tega, katero vrsto projekta smo izbrali, ko smo se lotili dela. Z gradniki se bomo zelo izčrpno ukvarjali v nadaljevanju.

NAMIG Gradnike lahko po želji dodajamo in odvzemamo. Vsak gradnik, ki ga lahko *naknadno* vključimo v projekt, je pravzaprav knjižnica ActiveX[1]. Seznam vseh napovedanih gradnikov v našem sistemu dobimo z ukazom **Components** iz priročnega menuja orodjarne. Če uporabimo več gradnikov, je smiselno orodjarno razdeliti na več logičnih enot. To omogoča ukaz priročnega menuja **Add Tab**, ki ga uporabimo pred dodajanjem novih gradnikov. Vsaka enota je ponazorjena z gumbom. Če želimo enoto odstraniti, moramo z desno tipko miške klikniti prav nad tem gumbom in iz priročnega menuja izbrati ukaz **Delete Tab**. Gumb ima namreč drugačen priročni menu kakor preostanek orodjarne.

Projektno okno

V projektnem oknu (*Project Explorer*) je seznam vseh odprtih projektov. Ta seznam se deli naprej na posamezne enote projekta, ki so lahko obrazci in programska koda[2]. Če bomo uporabljali privzeti vmesnik MDI (ki ga srčno priporočam), bomo projektno okno pogosto uporabljali. Z njim najlaže izbiramo enote projekta, ki jih zatem urejamo na delovni površini. V vrstici z gumbi, ki pripada projektnemu oknu, sta gumba za hiter preklop s kode na grafični vmesnik in nasprotno, ki ju pogosto uporabljamo pri načrtovanju obrazcev.



Slika 2-4: Projektno okno prikazuje vse odprte projekte in njihove sestavne enote. Ko jih želimo prikazati, si pomagamo z gumboma v vrhnji vrstici.

NAMIG V visual basicu se lahko hkrati ukvarjamo z več odprtimi projekti. Če želimo ob obstoječem odpreti še kakega, v glavnem menuju **File** izberemo ukaz **Add Project...** Ko je odprtih več projektov,

[1] Poudarjam *naknadno*, ker je začetni nabor gradnikov sestavljen tudi iz elementov, ki niso knjižnice ActiveX, temveč so del izvajalne knjižnice, ki pripada vsakemu programu, pisanemu v visual basicu (*intrinsic controls*).

[2] Kot smo že omenili v uvodu tega dela, predmetne narave visual basica zaenkrat še ne bomo obravnavali. V splošnem velja, da so v projektnem oknu lahko našteti tudi predmetni moduli in še nekaj vrst enot projekta.

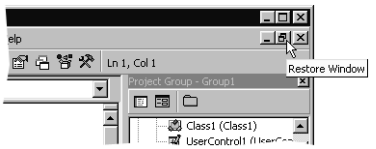
se vedno prevaja in zaganja samo eden. Ta je v projektnem oknu označen s krepko izpisanim imenom. Iz priročnega menuja vsakega imena projekta lahko izberemo ukaz **Set as Start Up** in ga s tem določimo kot zagonskega.

Načrtovanje obrazca

Največji del zaslona obsega prostor za načrtovanje obrazca (*Form Designer*). To je posledica privzete nastavitve, ki delovno okno raztegne čez ves prostor, ki je ostal na voljo znotraj vmesnika MDI. Če želimo, lahko tudi znotraj te oblike vmesnika prikazujemo več odprtih delovnih površin, ki so lahko bodisi okna Form Designer bodisi programerski urejevalniki, v katerih nastaja programska koda. Pri projektih z veliko enotami se rado zgodi, da postane vmesnik prenatrpan z različnimi okni, zato je včasih najbolje pustiti delovno okno raztegnjeno, tako da vselej vidimo samo eno enoto in delamo z njo.

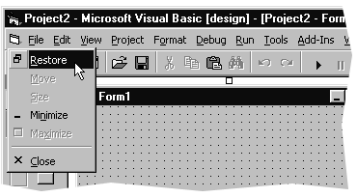
Če želite delovno okno pomanjšati, izberite ustrezen gumb iz naslovne vrstice podrejenega okna (*child window*). Vidite ga tudi na spodnji sliki:

Slika 2-5: Pomanjšanje okna z gumbom iz naslovne vrstice



Enak učinek dosežete tudi, če v nadzornem menuju (*system menu*) podrejenega okna izberete ukaz **Restore**:

Slika 2-6: Pomanjšanje okna z ukazom nadzornega menuja



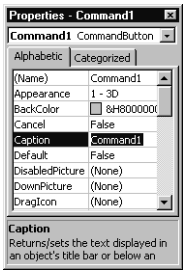
V prvotno stanje se vrnemo tako, da okno *povečamo* (*maksimiziramo*). To najlaže storimo, če dvakrat kliknemo naslovno vrstico okna. Na voljo sta tudi ukaz iz nadzornega menuja (**Maximize**) oz. gumb, v katerega se je spremenil junak s predzadnje slike.

Nova delovna okna odpiramo s projektnim oknom. Ko imamo na delovni površini že zbranih več oken – raztegnjenih ali ne –, z enega na drugega najenostavneje preklapljamo z menujem **Window**, v katerem so odprta okna našteta pri dnu. Na voljo je tudi priročna kombinacija tipk: Ctrl+Tab.

Lastnosti

Pri načrtovanju vmesnika določamo videz oken, s katerimi se bo ukvarjal uporabnik našega programa. To načrtovanje je sestavljeno iz risanja izbranih elementov in nastavljanja njihovih *lastnosti*, kot je npr. napis ali barva elementa. Lastnosti nastavljamo v posebnem oknu, *Properties*. Ogledate si ga lahko na spodnji sliki:

Slika 2-7: Okno za nastavljanje lastnosti



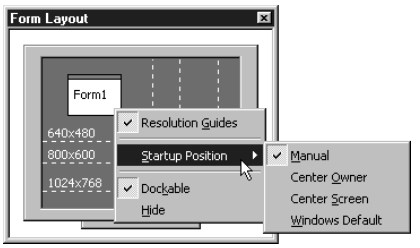
Okno je sestavljeno iz seznama vseh elementov, ki so na voljo, tabele lastnosti izbranega elementa in krajšega opisa izbrane lastnosti. Opis lahko po želji tudi izključimo s priročnim menujem (ukaz **Description**). S tem pridobimo nekaj prostora. Tabela lastnosti je lahko urejena abecedno ali pomensko. Eno ali drugo izbiramo z jezičkoma na vrhu tabele.

Umestitev okna

Pri uporabi vmesnika MDI opazimo še eno značilnost, ki je vmesnik SDI ne pozna. Ker obrazec načrtujemo v podrejenih oknih, položaj tega okna ne odraža položaja načrtovanega okna na zaslonu, ko bomo program pognali. Pri vmesniku SDI je položaj okna med načrtovanjem hkrati tudi začetni položaj okna med izvajanjem.

Položaj okna lahko pri obeh vmesnikih nastavimo z ustreznimi lastnostmi, na voljo pa je tudi posebno okno, *Form Layout*. Okno ste lahko opazili že na prvi sliki uporabniškega vmesnika, tu spodaj pa je prikazano bolj natančno:

Slika 2-8: Okno Form Layout omogoča nastavitev izhodiščnega položaja okna pri izvajanju programa.



Priročni menu ponuja nekaj ukazov, ki olajšajo postavitev okna. Okno lahko tudi zgrabimo z miško in ga odvlečemo na poljubno mesto. Koristno je vklopiti mrežo

ločljivosti (*Resolution guide*), ki nakazuje osnovne ločljivosti zaslona. Tako lahko tudi pri večji delovni ločljivosti načrtujemo okna, ki se bodo prikazovala na manj zmogljivih zaslonih, in nasprotno.

> **NAMIG** Če izberemo položaj na sredini zaslona (**Center Screen**), se bo sredina zaslona samodejno izračunala ob zagonu programa, in to pri vsaki ločljivosti. Dejanski položaj okna je zato odvisen od trenutne ločljivosti. Položaj **Windows Default** zadolži operacijski sistem, da poskrbi za začetni položaj okna. Okno se lahko ob vsakem zagonu izriše kje drugje.

Prilagajanje vmesnika

Uporabniški vmesnik razvojnega okolja visual basica si je mogoče temeljito prikrojiti po lastnih željah in potrebah. Začetnikom tega resda ne priporočam, saj se bodo v tej knjigi teže znašli, če bodo okolje preuredili. Povsem drugače pa je z izkušenimi uporabniki, ki jih lahko razporeditev menujev, gumbov in oken moti pri učinkovitem delu.

Ker se z videzom vmesnika razvojnega okolja v nadaljevanju ne bomo več ukvarjali, si oglejmo še nekaj možnosti, kako si lahko vmesnik prilagodimo, ki jih doslej še nismo omenili. Tematika, ki jo bomo obravnavali tu, malce presega doslej pridobljeno znanje, zato manj izkušenim uporabnikom priporočam, naj si spodnje nasvete o prilagajanju vmesnika preberejo pozneje.

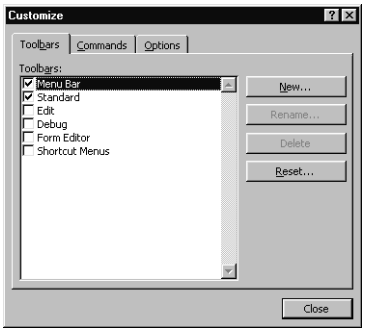
Preurejanje vrstic

Glavni menu je v visual basicu zgolj še ena vrstica z »gumbi«. Nad njim ali nad katerokoli drugo vrstico z gumbi lahko, če kliknemo z desno tipko miške, prikličemo priročni menu, ki smo ga že spoznali pri prvi obravnavi priročnih menujev zgoraj. Z njim lahko hitro vklapljamo in izklapljamo različne vrstice z gumbi. No, tu nas zanima njegov najzmogljivejši ukaz, ukaz **Customize**. Če ga izberemo, nas pozdravi pogovorno okno (*dialog box*) za prilagoditev vrstic z gumbi (slika 2-9).

> **NAMIG** Enak menu lahko izberemo tudi z izbiro **Toolbars** iz glavnega menuja **View**.

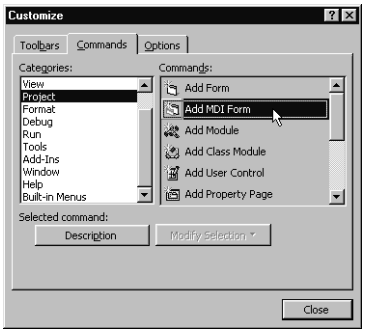
Kakor vidimo, lahko na prvi strani tega pogovornega okna pregledno vklapljamo in izklapljamo še več vrstic z gumbi, med drugim tudi glavni menu in menu bližnjic (*Shortcut Menus*).

Slika 2-9: Prilagajanje vrstic z gumbi



Na naslednji strani, ki se skriva pod jezičkom **Commands**, je seznam vseh ukazov, ki jih poznajo menuji in gumbi razvojnega okolja:

Slika 2-10: Seznam vseh ukazov, ki so na voljo



Ta stran nam omogoča, da preuredimo vse prikazane vrstice z gumbi. Če želimo v vrstico z gumbi dodati nov ukaz, najprej z levega seznama izberemo skupino ukazov, ki mu pripada, zatem pa z desnega seznama izberemo želeni ukaz in ga z miško odvlečemo na poljubno mesto v poljubni vrstici z gumbi.

Poljuben gumb odstranimo tako, da ga z miško zvlečemo iz vrstice z gumbi in izpustimo. Če ga bomo izpustili nad drugo vrstico z gumbi ali kje drugje nad isto vrstico, se bo zgolj preselil tjakaj.

Še večjo prilagodljivost omogoča gumb **Modify Selection**, ki je dejaven samo ob izbranem gumbu v kateri izmed vrstic (gumb izberemo, ko ga enkrat kliknemo z levim gumbom miške).

NAMIG Če ga pri preurejanju gumbov polomite, se vrnite na prvo stran pogovornega okna (jeziček **Toolbars**), s seznama izberite želeno vrstico z gumbi ter kliknite gumba **Reset** in **Application that sent this**.

Gumbe lahko preurejate tudi brez prikazanega pogovornega okna **Customize**. Prestavljate in brišete jih tako, da najprej pritisnete tipko Alt, nato pa jih z miško odvlečete na novo mesto ali stran iz orodjarne.

Preurejanje oken

Za delo v visual basicu je priporočljivo uporabljati vmesnik MDI, ki omogoča hkratno delo z več odprtimi dokumenti, kar so v našem primeru okna z grafičnim vmesnikom in programsko kodo. Druga okna – pravimo jih tudi pomožna – lahko po želji skrivamo in preurejamo na vmesniku, najbolj zanimiva pa je možnost *sidranja*. Sidranje (*docking*) postavi pomožno okno v raztegljiv okvir ob eni izmed stranic glavnega okna. Sidramo lahko skoraj vsa pomožna okna.

NAMIG Okna, ki jih je mogoče sidrati, določamo s stikali, ki se skrivajo v naslednjem pogovornem oknu: **Tools ➪ Options**, jeziček **Docking**.

Vsa razpoložljiva pomožna okna so našteta v glavnem menuju View. Večina pomožnih oken je po privzeti nastavitvi že prikazanih, odpremo pa lahko tudi nova, kot sta npr. paleta barv (Color Palette) ali izvršilno okno (Immediate Window). Pomožna okna, ki jih lahko sidramo, prepoznamo po posebni naslovni vrstici:

Slika 2-11: Naslovna vrstica okna odkriva, da ga je moč zasidrati.

V naslovni vrstici ni gumba nadzornega menuja, skrajno desno je le gumb za njegovo skrivanje. Z vlečenjem naslovne vrstice okno prestavimo na novo mesto, ki ga nakazuje njegov obris. Če okno odvlečemo v bližino ene od stranic glavnega okna, se obris rahlo spremeni in nakaže zasidrani položaj.

NAMIG Sidranje preprečimo tako, da med vlečenjem tiščimo pritisnjeno tipko Ctrl.

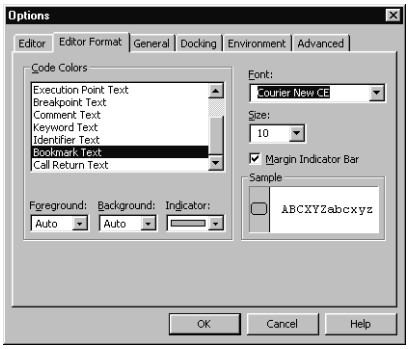
Sidramo lahko tudi vse vrstice z gumbi.

Okno s programsko kodo

Oglejmo si še zelo pomembne nastavitve, ki vplivajo na videz in vedenje okna s programsko kodo. Tega okna doslej še nismo uporabljali, vendar smo tik pred tem.

Iskane nastavitve se skrivajo v pogovornem oknu, ki ga odpre ukaz **Options** iz glavnega menuja **Tools**. Z nastavitvami urejevalnika programske kode se ukvarja kar precej strani tega pogovornega okna, tu pa si oglejmo stran pod jezičkom **Editor Format**:

Slika 2-12: Nastavitve urejevalnika programske kode



Za nas je od nastavitev gotovo najbolj zanimiva izbor pisave. Če želimo, da bodo pravilno prikazani tudi nizi s slovenskimi znaki, izberimo s seznama **Font** pisavo »Courier New CE«[3].

Po želji lahko spremenimo še velikost znakov (seznam **Size**), vidnost označevalnega stolpca (stikalo **Margin Indicator Bar**) in barvo posameznih skladenjskih elementov programske kode. Menjave barv ne priporočam, ker se mi zdijo privzete barve dobro izbrane.

[3] Pri tem predpostavljam, da uporabljate slovenska Okna ali vseevropsko različico (*paneuropean*) s slovenskimi krajevnimi nastavitvami (*regional settings*).

3. POGLAVJE

Pozdravljen, svet

Pričujoče poglavje bo predstavilo razvoj prvega programa v visual basicu. Ne bo šlo zgolj za skupek napotkov, ampak si bomo stvari ogledali bolj temeljito. Visual basic je bil eden prvih izdelkov, ki so podpirali vizualno programiranje. Z njim lahko programe razvijamo precej hitreje in z manj ročnega kodiranja, zato je manj tudi možnosti, da bi se prikradle napake. Besedilo v nadaljevanju bo še posebej zanimivo za tiste, ki v Oknih še niso programirali. V nadaljevanju bomo deležni tudi precej namigov, kako si olajšati pogosta opravila. Zato vsem priporočam, da si to poglavje vsaj bežno preberete, bolj izkušeni pa lahko preostanek prvega dela tudi preskočijo. Snidemo se spet v naslednjem poglavju, ki visual basic obravnava sistematično.

Nastajanje prvega programa

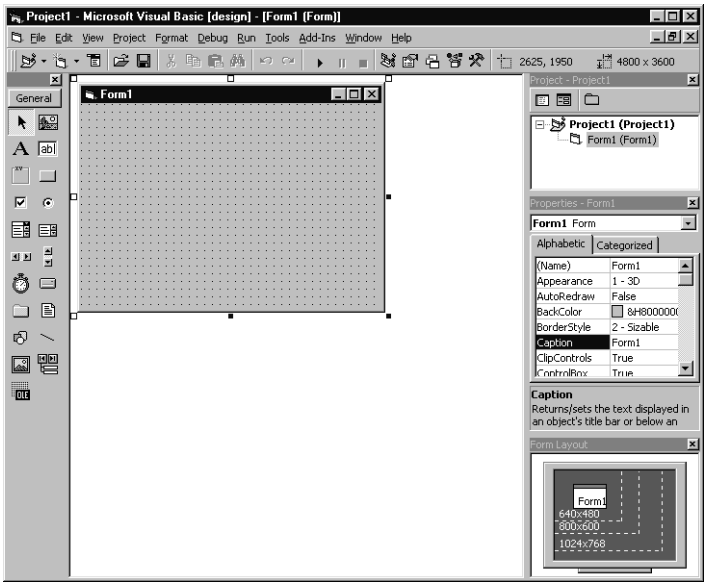
Po navodilih, ki sledijo, bomo sestavili naš prvi program v visual basicu. Spoznali bomo osnovne tehnike, ki omogočajo vizualno programiranje, in deležni bomo nekaj namigov, kako si olajšati delo. Osnove vizualnega programiranja pridejo na vrsto pozneje, ko bomo imeli vsaj približno občutek, kako vse skupaj deluje.

Risanje vmesnika

Pri vizualnem programiranju je navadno prvi korak kar risanje uporabniškega vmesnika. V tej fazi načrtujemo videz programa, kakor ga bo pozneje videl uporabnik. Delo je podobno sestavljanju sličic, zato se visual basic kmalu prikupi tudi uporabnikom, ki jih je strah že ob sami misli na programiranje. V Oknih je uporabniški vmesnik sestavljen iz več, hmmm, oken (presenetljivo, kaj?). V visual basicu se ta okna imenujejo *obrazci* (*forms*). Če ob zagonu visual basica z uvodnega zaslona izberemo privzeto možnost (**Standard EXE**), je uporabniški vmesnik že pripravljen za načrtovanje videza prvega obrazca. Največ prostora zaseda okno *Form Designer*, kar je razvidno tudi s slike 3-1.

Pred načrtovanjem obrazca moramo seveda imeti vsaj približno predstavo o tem, kaj naj bi naš program sploh počel. No, za začetek naj ne postori nič drugega, kakor da nas pozdravi. Čeprav bi to lahko storil že takoj po zagonu programa, raje uporabimo neki drug sprožilni mehanizem. Tudi v drugih okenskih programih se veliko operacij zgodi, ko z miško kliknemo gumb. Naj bo tako tudi tu. Potrebujemo torej gumb.

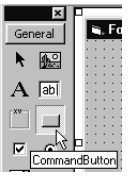
Slika 3-1: Uporabniški vmesnik visual basica pred načrtovanjem prvega obrazca



Dodajanje elementa

Nov element na vmesnik narišemo tako, da ga najprej izberemo v orodjarni (*Toolbar*). Kliknemo sličico želenega gradnika. Ker potrebujemo *gumb*, izberimo gradnik *CommandButton*, ki ga vidite na sliki.

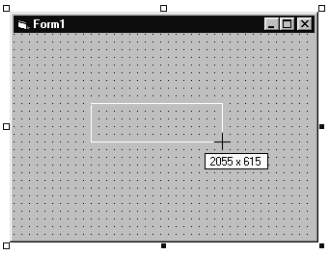
Slika 3-2: Gumb, ki ga potrebujemo, se skriva tule.



Ko izberemo želeni element, se z miško premaknemo nad obrazec. Kazalec miške se spremeni v križ (*crosshair*), kar pomeni, da lahko izbrani element tu nekje izrišemo. Izrišemo ga tako, da z miško pokažemo na izhodiščni položaj, nato pa miško s pritisnjenim gumbom vlečemo toliko časa, dokler element ne doseže želene velikosti. Med risanjem je bodoči element samo nakazan. Če med risanjem za hip obmirujemo, se tik ob kazalcu miške prikaže okence s trenutnimi merami elementa. Enake mere lahko opazujemo tudi skrajno levo v vrstici z gumbi zgoraj. Poizkusimo narisati gumb, približno podoben temu na sliki 3-3.

Ko gumb miške izpustimo, se gradnik dokončno izriše. Dobro se je ozreti tudi po orodjarni. Ko smo element izrisali, njegova slika ni več izbrana! Če želimo narisati drug gumb ali katerikoli drug element, ga moramo vnovič izbrati v orodjarni.

Slika 3-3: Risanje gumba na vmesniku



NAMIG Če želimo zapored izrisati več enakih elementov, izberemo sličico v orodjarni tako, da ob kliku z miško tiščimo tipko Ctrl na tipkovnici. Tako bo ostal izbrani element vključen, dokler ne izberemo drugega.

Element privzete velikosti na privzetem mestu hitro izrišemo, če dvakrat kliknemo njegovo sličico v orodjarni.

Preurejanje

Seveda nam ni treba že v prvem poizkusu izrisati vmesnika takega, kakor si želimo. Za začetek zadošča, da nekam na obrazec namečemo želene elemente. Vsakega izmed njih lahko prestavljamo, povečamo ali pomanjšamo in ga na koncu tudi odstranimo, če nam ne koristi več.

Zelo pomembno: elemente je mogoče preurejati le, kadar je v orodjarni izbran kazalec miške (*Pointer*). Predstavlja ga gumb desno zgoraj, ki ni simbol nobenega elementa vmesnika, ampak orodje za preurejanje. Če izbiramo elemente na ustaljeni način, visual basic samodejno izbere kazalec, brž ko je risanje končano. Drugače je, če smo element izbrali in hkrati pritisnili na tipko Ctrl (glej zgornji namig). Potem moramo kazalec posebej izbrati.

Slika 3-4: Kazalec, ki omogoča preurejanje elementov na vmesniku

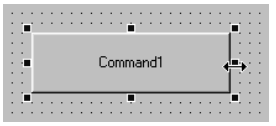
Nazadnje izrisani element je obdan z osmimi kvadratki, *ročicami* (*handles*), s katerimi lahko spreminjamo njegovo velikost. Ročice nakazujejo trenutno izbrani element. Če želimo preurejati kakega drugega, kliknemo njegovo površino.

Čeprav smo izrisali samo en gumb, imamo že opraviti z dvema elementoma. Eden izmed njiju je vedno tudi sam *obrazec*, na katerega rišemo druge elemente. To hitro

preverimo tako, da z miško kliknemo njegovo površino (ki je ne prekriva gumb). Ročice se prikažejo ob robovih obrazca. Če gumb znova kliknemo, se ročice prestavijo nazaj obenj.

Velikost izbranega elementa spremenimo tako, da z miško pokažemo na ročico. Njen kazalec se spremeni in zdaj lahko z vlečenjem popravljamo izbrano dimenzijo. Položaj izbranega elementa spremenimo preprosto tako, da ga z miško odvlečemo drugam. Prej moramo z miško pokazati nekam v njegovo notranjost (kazalec miške ostane navadne oblike).

Slika 3-5: Z ročicami spreminjamo velikost elementa



NAMIG Izbrani element lahko prestavljamo in raztegujemo tudi s tipkovnico. Če tiščimo pritisnjeno tipko Ctrl in hkrati pritiskamo na smerne tipke (puščice levo, desno, gor in dol), spreminjamo njegov položaj, če pa tiščimo pritisnjeno tipko Shift, se spreminja njegova velikost.

Če želimo spremeniti položaj obrazca, bomo imeli pri vmesniku MDI nekaj težav. Pet ročic je namreč votlih, kar ponazarja njihovo neuporabnost. Spremenimo lahko širino in višino, ne pa tudi položaja. Kakor smo že spoznali pri opisu razvojnega okolja visual basica, imamo pri vmesniku MDI na voljo posebno okno (*Form Layout*), ki omogoča, da določamo izhodiščni položaj obrazca.

Odvečni element zbrišemo z ukazom iz priročnega menuja. Če ga kliknemo z desno tipko miške, se prikaže priročni menu. Ta se pri obrazcu nekoliko razlikuje od priročnega menuja vseh drugih gradnikov. Obrazca npr. ne moremo tako preprosto odstraniti. V priročnem menuju drugih gradnikov bomo na koncu prve skupine ukazov našli ukaz Delete, ki izbrani element odstrani. Izbrani element lahko zbrišemo tudi tako, da pritisnemo tipko Del.

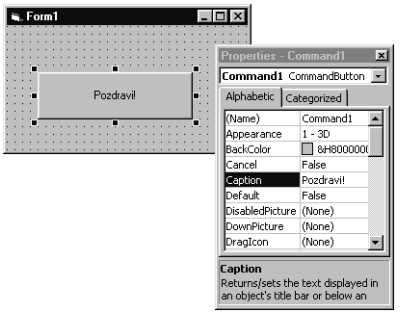
Nastavljanje lastnosti

Načrtovanje vmesnika je sestavljeno iz risanja in nastavljanja *lastnosti* (*properties*), ki dokončno določi videz uporabniškega vmesnika. Nekatere lastnosti smo spreminjali že z risanjem. To so lastnosti, ki določajo položaj in velikost posameznega elementa. Mnogo lastnosti pa ni mogoče določiti zgolj z risanjem, npr. napis na gumbu. Vse lastnosti elementa so zato zbrane v posebnem oknu, v katerem jih lahko pregledujemo in popravljamo. Pomembno je, da imajo vse neko privzeto nastavitev, tako da je ponavadi treba naknadno določiti zgolj malo nastavitev. Pri gumbu večinoma spremenimo le napis.

Okno za nastavljanje lastnosti (*Properties*) smo bežno že spoznali pri opisu uporabniškega vmesnika razvojnega okolja visual basica. Naj poudarim, da okno s tabelo lastnosti vedno odraža nastavitve trenutno izbranega elementa. To je natančno tisti element, ki je na obrazcu obdan z ročicami. Okno Properties ima pri vrhu zelo priročen seznam, s katerega lahko izberemo drug element, ne da bi se vračali k obrazcu. Na tem seznamu so vedno našteti vsi elementi obrazca, s katerim se ukvarjamo. Včasih so posamezni elementi zakriti in jih z miško sploh ne moremo izbrati. Takrat je ta seznam nadvse uporaben.

Pred nadaljevanjem preverimo, ali je na seznamu (in zato tudi na obrazcu) izbran element »**Command1** CommandButton«. Sedaj lahko nastavimo katero izmed njegovih lastnosti. Zanima nas napis na gumbu, ki določa lastnosti, Caption. Želeno lastnost najdemo v tabeli in jo spremenimo preprosto tako, da popravimo ustrezno vrstico. Napis »Command1« popravimo v »Pozdravi!«.

Slika 3-6: Ko popravimo lastnost, je gumb bolj uporaben.



Za naš preprost zgled je to vse. Naj omenim še lastnosti Left, Top, Width in Height, ki določajo položaj in velikost elementa (po vrsti levi zgornji vogal izhodišča ter širino in višino). Lastnosti, ki jih posamezen element pozna, so od primera do primera različne, štiri prej naštete pa ima vsak vidni element (lastnosti Left in Top poznajo tudi nevidni).

NAMIG Okno za nastavljanje lastnosti sprožimo s priročnim menujem vsakega elementa na obrazcu (ukaz **Properties**), uporabimo pa lahko tudi bližnjico s tipkami: kombinacija Ctrl+Alt.

Zelo pomembno: vsak element ima prav gotovo še eno lastnost – ime (lastnost Name). Ime elementa lahko uporabimo v programski kodi (se nanj sklicujemo), zato mora imeti vsak element na obrazcu *enoličnо* določeno ime. Visual basic sam skrbi, da imajo vsi elementi drugačno, privzeto ime. V preprostih zgledih, med kakršne sodi tudi ta naš programček, imena elementa *ne spreminjamo*. Če pa ga spremenimo, moramo natančno vedeti, kaj počnemo, ker lahko zaradi določenih nerodnosti v razvojnem okolju visual basica naredimo več škode kakor koristi. Zaenkrat pustimo

lastnost Name tako, kakršna je po privzeti nastavitvi. Ker je tako zelo pomembna, je v oknu za nastavljanje lastnosti označena kot »(Name)«, torej v oklepajih. Zato jo vedno najdemo na vrhu seznama vseh lastnosti (če so te urejene po abecednem redu).

Pisanje kode

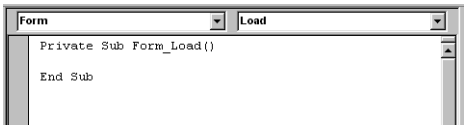
Risanje elementov in nastavljanje njihovih lastnosti določi zunanji *videz* programa. Kako se bo ta program *vedel*, pa je mogoče določiti samo s pisanjem programske kode. Za naš preprost zgled bo treba zapisati samo eno vrstico, pa še pri tej nam bo visual basic precej pomagal.

Vsakemu obrazcu pripada tudi okno, ki omogoča pisanje in urejanje programske kode (*Code*). To okno na začetku ni vidno, lahko pa ga prikličemo na precej različnih načinov:

- V priročnem menuju elementa izberemo ukaz **View Code**.
- V projektnem oknu izberemo želeni obrazec in kliknemo gumb **View Code** ali izberemo istoimenski ukaz priročnega menuja.
- V glavnem menuju **View** izberemo ukaz **Code**.
- Najbolj priročno pa je pritisniti na tipko F7.

> **NAMIG** Okno s kodo lahko odpremo tudi tako, da izbrani element dvakrat kliknemo z miško. V tem primeru se kazalec vnosne točke že postavi v značilni odzivni podprogram za izbrani element.

Videz okna s programsko kodo si lahko ogledate na spodnji sliki:



Slika 3-7: Okno za razvoj programske kode

Sestavni deli zgornjega okna sta dva seznama in urejevalnik programske kode. Seznama sta zelo pomembna. Z njima najdemo pravo mesto v programski kodi, na katero sodi naš programski stavek.

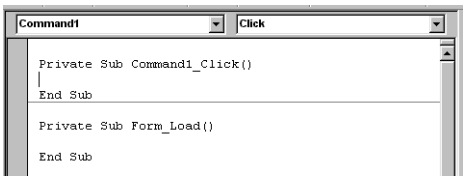
Levi seznam (imenovan *Object*) ponuja vse elemente, ki so postavljeni na obrazec, ter sam obrazec. Potem je seznam precej podoben tistemu v oknu za nastavljanje lastnosti. Vseeno je med njima nekaj razlik. Na seznamu okna s kodo so elementi našteti le po imenu in ne tudi po vrsti oziroma tipu (to je precejšnja pomanjkljivost). Obrazec je vedno ponazorjen z imenom **Form** in ne z dejanskim imenom (lastnostjo

Name), ker ima vsak obrazec svoje okno s kodo. Pri vrhu seznama je še vrstica **(General)**, ki ponazarja splošno kodo, ki ni neposredno vezana na noben izrisani element.

Vsebina desnega seznama (imenovanega *Procedures*) se spreminja glede na izbrano vrstico levega seznama. Če je na levem seznamu izbran kateri izmed elementov uporabniškega vmesnika, lahko izbiramo med *odzivnimi* podprogrami. Če je levo izbran splošni del (*general*), pa lahko izbiramo med napovedmi (*declarations*) in morebiti definiranimi *splošnimi* podprogrami.

Za naš prvi zgled moramo določiti zelo preprosto vedenje programa. Ko uporabnik klikne gumb, naj nas program pozdravi. Napisati moramo *odzivni* podprogram za *dogodek* »klik gumba«. To dosežemo tako, da z levega seznama izberemo ukazni gumb, ki se skriva pod privzetim imenom **Command1**, z desnega seznama pa izberemo odzivni podprogram **Click** (verjetno bo izbran že samodejno). Okno s programsko kodo mora biti sedaj takole:

Slika 3-8: Okno s programsko kodo je sedaj pripravljeno za našo programsko vrstico.



V podprogram `Private Sub Command1_Click` bomo vnesli naslednji programski stavek:

```
MsgBox "Pozdravljen, svet!"
```

Pred vnosom se prepričajmo, ali je kazalec vnosne točke (*caret*) na pravem mestu, med stavkoma `Private Sub Command1_Click` in `End Sub` (kot na sliki). Lepo bi bilo, ko bi programsko kodo malo zamaknili, zato najprej pritisnemo tipko Tab. Zapišemo ime *vgrajenega* podprograma: `MsgBox`. Brž ko vnesemo še presledek, nas visual basic preseneti:

Slika 3-9: Dobrodošla pomoč pri kodiranju

Pozdravi nas pripomoček, ki izdatno olajša kodiranje. V rumenem okencu nas pozdravi seznam vseh *parametrov*, ki jih pravkar izpisani podprogram potrebuje.

Pripomoček je še posebej uporaben pri bolj eksotičnih podprogramih, ki jih redkeje uporabljamo. Prihrani nam marsikateri vpogled v pomoč ali priročnike, saj lahko zelo hitro ugotovimo, kaj vse uporabljeni podprogram potrebuje, da bo lahko dokončal svoje delo. Mi lahko nemoteno tipkamo naprej, saj že vemo, kaj želimo. Izpisana programska koda naj bo videti takole:

Slika 3-10: Dokončan program

Iz okna s kodo se lahko kadarkoli vrnemo nazaj k risanju vmesnika. Možnosti so naslednje:

- V projektnem oknu izberemo želeni obrazec in kliknemo gumb **View Code** ali izberemo istoimenski ukaz priročnega menuja.
- V glavnem menuju **View** izberemo ukaz **Object**.
- V glavnem menuju **Windows** pri dnu najdemo našteta vsa odprta podrejena okna, med drugim tudi **Form1 (Form)**.
- Najbolj priročen pa je pritisk na kombinacijo tipk Shift+F7.

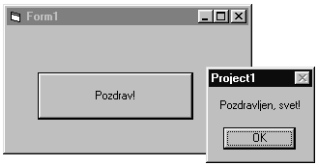
Preizkus

Tako, zgled smo dokončali. Preostane nam samo še, da ga preizkusimo. S tem preidemo z načrtovanja (*design time*) na izvajanje (*runtime*). Če je s programom vse v redu, nas bo, ko bomo kliknili gumb, pozdravil s sporočilom. Zagon programa dosežemo na več načinov:

- V glavnem menuju **Run** izberemo ukaz **Start**.
- Kliknemo gumb Start v vrstici z gumbi pod glavnim menujem.
- Najhitreje pa gre, če pritisnemo na tipko F5.

Če ni napak, nas pozdravi glavno okno našega programa (zdaj mu ne rečemo več obrazec, saj smo prešli na izvajanje). Če kliknemo gumb, lahko uživamo v delu podprograma MsgBox. Ta prikaže novo okno s pozdravnim besedilom, ki ga zapremo, ko kliknemo gumb OK.

Čestitam! Tudi če to ni vaš prvi program v visual basicu. Če se vam je kje zalomilo, preverite postopek od začetka, tu pa sta še dve opozorili:



Slika 3-11: Naš prvi zgled med izvajanjem

- Preverite programski stavek, ki kliče podprogram MsgBox. Je besedilo med narekovajema?
- Če je stavek v redu, preverite, ali je v pravem odzivnem podprogramu!

> **NAMIG** Če program še vedno ne deluje, ga, tako kakor vse druge zglede iz knjige, najdete na priloženi plošči CD-ROM.

Hm, program deluje, toda nekako ga je treba še zaustaviti. Čeprav nismo predvideli nobenega programskega mehanizma, je na voljo kar nekaj načinov zaustavljanja izvajanja:

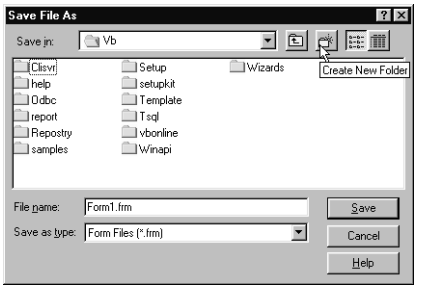
- Kliknemo gumb za zapiranje okna. To lahko storimo le tedaj, ko ni prikazano sporočilo.
- Preklopimo v visual basic in v menuju **Start** izberemo ukaz **End**.
- V vrstici z gumbi visual basica kliknemo gumb End.
- Gre tudi s kombinacijo tipk Alt+F4.

> **NAMIG** Visual basic po privzeti nastavitvi program prevaja postopoma. To pomeni, da se najprej prevedejo le podprogrami, nujni za zagon. Če programa ne preizkusimo v celoti, celo obstaja možnost, da se določeni podprogrami ne bodo nikoli uporabili in zaradi tega tudi nikoli ne prevedli.
>
> Postopno prevajanje ima še eno pomanjkljivost. Preproste, skladenjske napake, ki bi jih prevajalnik lahko našel, ostanejo ob zagonu programa neodkrite in nas neugodno presenetijo kot izvajalne napake.
>
> Če se želimo tem težavam izogniti, lahko postopno prevajanje izključimo. V glavnem menuju **Tools** izberemo ukaz **Options...** V prikazanem pogovornem oknu izberemo jeziček **General** in v skupini **Compile** izklopimo stikalo **Compile On Demand**.

Shranjevanje

Vsekakor je čas, da projekt shranimo na disk. Vsak projekt v visual basicu je sestavljen iz več datotek, zato je priporočljivo, da ga shranimo v posebno mapo. Na

srečo je novo mapo mogoče ustvariti tudi tik pred shranjevanjem. Ves projekt shranimo z ukazom **Save Project...**, ki se skriva v glavnem menuju **File**. Na voljo je tudi enakovreden gumb v vrstici z gumbi. Priporočam torej, da pred shranjevanjem ustvarite novo mapo (*folder*), jo odprete in vanjo shranite vse datoteke projekta.



Slika 3-12: V pogovornem oknu za shranjevanje datotek je mogoče ustvariti tudi novo mapo.

Najprej shranimo datoteko, ki opisuje videz obrazca. Njeno privzeto ime je »Form1.frm«. Ime lahko po želji spremenimo, ni pa nujno. Na koncu moramo shraniti še datoteko, ki opisuje vse sestavne dele projekta, *projektno datoteko*. Njeno privzeto ime je »Project1.vbp«. Priporočam, da ime spremenite v kaj zgovornejšega, npr. »Prva vaja«. Končnice ni treba dopisovati, ker se doda samodejno.

NAMIG Visual basicu lahko naložimo, da pred vsakim zagonom projekta samodejno shrani vse spremenjene datoteke. Tovrstna nastavitev ima dobre in slabe strani. Dobra je vsekakor ta, da nam ni treba skrbeti za shranjevanje sprememb in je projekt, tudi če se sesuje izvajalno okolje, varno shranjen. Slabost pa je v tem, da samodejno shranjevanje prekrije prejšnjo različico, zato ne moremo imeti hkrati na disku stare in popravljene različice (več o shranjevanju pod novim imenom si lahko preberete v namigu spodaj, pod naslovom *Dodajanje kode*). Ta težava ni tako huda, saj za posodabljanje pripravljenih zgledov raje uporabljamo mehanizem *predlog* (*templates*).

Samodejno shranjevanje ob zagonu omogočimo z naslednjim ukazom: **Tools ➪ Options ➪** jeziček **Environment ➪** stikalo **Save Changes** v okvirčku When a program starts. Na izbiro imamo tudi stikalo **Prompt To Save Changes**, ki nas vedno znova vpraša, ali želimo spremembe shraniti.

Dopolnitev zgleda

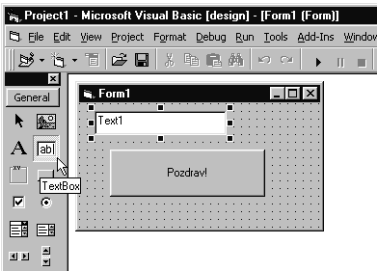
Naš prvi zgled bomo še malo dopolnili, tako da bomo spoznali še nekatere tehnike načrtovanja vmesnika. Razširimo naloge našega programa. Namesto neosebnega

pozdrava naj uporabnika raje pozdravi po imenu. Da bo to mogoče, moramo uporabniku omogočiti, da vnese svoje ime. Ponavadi za vnašanje uporabimo poseben gradnik, čeprav imamo za preprosto vnašanje na voljo tudi podprogram.

Urejanje več gradnikov

Na uporabniški vmesnik bomo dodali nov element, gradnik *vnosno polje* (Text Box). Izberemo njegovo sličico v orodjarni in ga narišemo na uporabniški vmesnik, ki naj bo zdaj približno tak, kakor na spodnji sliki:

Slika 3-13: Dodano vnosno polje

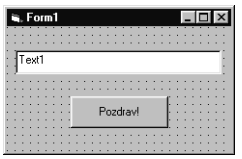


Namenoma želim, da bi se ne ukvarjali preveč s poravnavanjem elementov. Visual basic ima posebna orodja, ki to postorijo namesto nas. Vendar si najprej oglejmo, kako lahko izberemo več elementov. Če kliknemo površino, ki jo pokriva določen element, ga izberemo. Če želimo sedaj hkrati izbrati še več elementov, pritisnemo in tiščimo tipko Ctrl ter klikamo druge elemente. Če kliknemo že izbranega, s tem prekličemo njegovo označitev. Vsi izbrani elementi so obdani z ročicami.

> **NAMIG** Več elementov lahko hitro izberemo tudi z *označevalcem* (*marquee*). Pri njegovi rabi moramo paziti, da z miško pokažemo nekam na še neuporabljeno površino obrazca. Če zatem s pritisnjenim gumbom vlečemo miško, označimo vse elemente, ki jih seka navidezni okvir označevalca, ki se med vlečenjem izrisuje.

Med izbiranjem več elementov lahko opazujemo okno za nastavljanje lastnosti (*Properties*). Vidimo, kako se število elementov v tabeli zmanjšuje. V njej so samo še tiste lastnosti, ki so *skupne* vsem izbranim elementom. Kako lahko hitro poravnamo levi rob vseh izbranih gradnikov? Enostavno nastavimo lastnost Left, npr. na 120. Spremenila se bo hkrati pri vseh izbranih elementih. Podobno jim lahko nastavimo enako širino zgolj z nastavitvijo lastnosti Width npr. na 3240.

Kaj pa, če bi želeli enak vmesnik kakor na sliki 3-14?



Slika 3-14: Poravnana elementa na obrazcu

Elementa sta med seboj in glede na obrazec poravnana na sredino. Za poravnavanje elementov imamo na voljo ukaze glavnega menuja **Format** in celo posebno vrstico z gumbi, imenovano Form Editor. Ukazi v glavnem delujejo samo, ko izberemo več elementov, njihov pomen pa je razložen spodaj:

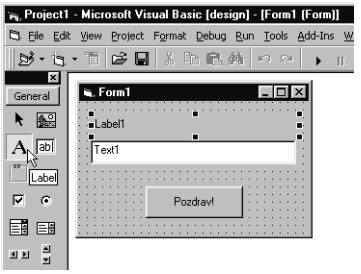
- Align
  - Lefts — Poravna leve strani izbranih elementov.
  - Centers — Poravna izbrane elemente na sredino vodoravno.
  - Rights — Poravna desne strani izbranih elementov.
  - Tops — Poravna zgornji rob izbranih elementov.
  - Middles — Poravna izbrane elemente na sredino navpično.
  - Bottoms — Poravna spodnji rob izbranih elementov.
  - To Grid — Poravna elemente v mrežo.
- Make Same Size
  - Width — Izenači širino izbranih elementov.
  - Height — Izenači višino izbranih elementov.
  - Both — Izenači velikost izbranih elementov.
- Horizontal/Vertical Spacing
  - Make Equal — Izenači razmik med elementi.
  - Increase — Poveča razmik med elementi.
  - Decrease — Zmanjša razmik med elementi.
  - Remove — Odstrani razmik med elementi.
- Center In Form
  - Horizontally — Postavi izbrane elemente na sredino obrazca vodoravno.
  - Vertically — Postavi izbrane elemente na sredino obrazca navpično.
- Order
  - Bring To Front — Postavi izbrane elemente v ospredje.
  - Send To Back — Postavi izbrane elemente v ozadje.
- Lock Controls — Prepreči preurejanje vseh elementov na obrazcu.

Če se pri urejanju elementov na kakem obrazcu še posebej potrudimo, se splača zakleniti njihov položaj in velikost. To postori ukaz Lock Controls, ki deluje kot stikalo. Kdor se precej ukvarja z urejanjem, naj si vklopi vrstico z gumbi Form Editor.

NAMIG Pri načrtovanju obrazca nam je v veliko pomoč pripravljena mreža (*grid*). Njene nastavitev so dosegljive v pogovornem oknu, ki se prikaže po izbiri ukaza **Tools ➪ Options...** Izbrati moramo jeziček **General**. Nastavitve mreže so v okvirčku Form Grid Settings. Stikalo Show Grid določa, ali je mreža vidna, stikalo Align Controls to Grid pa določa, ali je dejavna. Po želji lahko nastavimo tudi lasten razmik med enotami mreže.

Vmesnik moramo dopolniti še z enim elementom. To je *napis* (Label), ki ga bomo dodali nad vnosno polje in bo pojasnjeval njegov namen. Izberemo njegovo sličico v orodjarni in ga narišemo na uporabniški vmesnik, ki naj bo sedaj približno tak kakor na spodnji sliki:

Slika 3-15: Dodan napis, ki bo pojasnil pomen vnosnega polja.



Da bo napis opravičil svoj smisel, mu popravimo lastnost Caption z »Label1« na »Vnesi svoje ime:«.

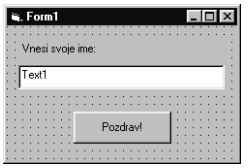
Dodajanje kode

Iz programske vrstice, ki smo jo zapisali v prvem zgledu, ni bilo razvidno, da lahko lastnosti gradnikov uporabljamo tudi v programski kodi. Vemo že, da ima vsak gradnik svoje enolično določeno ime (lastnost Name), ki omogoča njegovo rabo v kodi. Če vemo, da lahko lastnosti dosegamo tudi iz programske kode, se kar ponuja vprašanje, ali je smiselno lastnosti sploh nastavljati s posebnim oknom že med risanjem vmesnika? To početje je še bolj vprašljivo, če se zavedamo, da nekaterih, včasih zelo pomembnih lastnosti v tem oknu sploh ne moremo uporabiti, ampak so na voljo le znotraj programske kode!

V praksi se izkaže, da se med risanjem izplača nastaviti lastnosti, ki neposredno vplivajo na videz elementov, kot je npr. njihov položaj ali napis. Nesmiselno bi bilo spreminjati položaj vsakega posameznega gradnika s programsko kodo. No, včasih smo v to prisiljeni, če npr. želimo ohraniti sredinsko poravnavo tudi po tem, ko uporabnik spremeni velikost obrazca.

Zakaj tak uvod? Odločiti se moramo, ali bomo spremenili kako lastnost vnosnega polja, ki smo ga malo prej dodali. Vsekakor nas moti privzeto besedilo, saj si verjetno želimo, da bi bilo vnosno polje na začetku prazno. To bi lahko dosegli z nastavitvijo ustrezne lastnosti, vendar bomo raje posegli po programski kodi. Predvsem zato, ker bomo isto lastnost potrebovali tudi pri izpisovanju obvestila, saj želimo uporabnika pozdraviti po imenu. Hkrati bomo spoznali tudi podprogram, v katerem lahko popravimo lastnosti gradnikov, še preden se ti prikažejo.

Izrisani programski vmesnik bi moral biti po dosedanjem delu približno tak kakor na spodnji sliki:



Slika 3-16: Uporabniški vmesnik izboljšanega zgleda

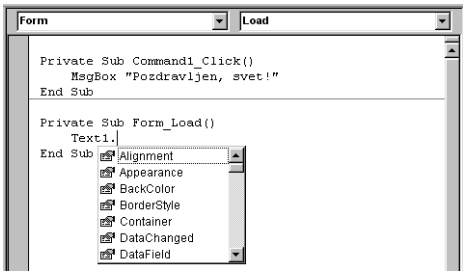
Zdaj nas čaka delo s programsko kodo. Na enega od prej opisanih načinov prikažemo okno s programsko kodo (najbolje, da pritisnemo tipko F7). Za začetek moramo zapisati podprogram, ki se nam že ves čas ponuja sam od sebe, `Private Sub Form_Load`. Ta je pravo mesto za nastavljanje lastnosti, ki bi jih radi posodobili, še preden se obrazec dejansko prikaže. Če kazalec vnosne točke slučajno ni v tem podprogramu, preverimo nastavitve na seznamih zgoraj. Na levem naj bo izbran element **Form**, na desnem pa podprogram **Load**.

V podprogramu je en sam programski stavek:

```
Text1.Text = ""
```

Njegov pomen je naslednji: elementu z imenom Text1 nastavi lastnost Text na prazen niz. Ali, z drugimi besedami, izprazni vsebino vnosnega polja.

Ko začnemo vnašati programsko kodo, nas visual basic znova prijetno preseneti, brž ko zapišemo `Text1.` (tudi piko).



Slika 3-17: Pomoč pri pisanju kode

Urejevalnik izvorne kode prepozna, da smo ravnokar izpisali ime gradnika in da se bomo sklicevali na neko njegovo lastnost ali *metodo* (sklicevanje določa posebni *operator* – pika). Zato nam nemudoma odpre seznam vseh lastnosti in metod, ki so zanj na voljo. Sedaj lahko zapišemo samo toliko črk, da postane želena lastnost izbrana (zadoščata T in E). Brž ko je izbrana lastnost Text, preprosto pritisnemo na preslednico in urejevalnik bo sam dokončal naziv lastnosti. Stvar je toliko bolj uporabna, kolikor daljše je ime lastnosti.

NAMIG Pomoč pri pisanju kode lahko vklapljamo ali izklapljamo s stikali, ki se skrivajo pod nastavitvijo: **Tools ➪ Options ➪** jeziček **Editor ➪** stikali **Auto List Members** (prikaz seznama lastnosti in metod) in **Auto Quick Info** (prikaz seznama parametrov podprograma).

Sedaj moramo posodobiti še programski stavek, ki nas čaka še iz prejšnjega zgleda. Sporočilo, ki ga izpiše podprogram MsgBox, bomo dopolnili z imenom uporabnika. Programski stavek se sedaj glasi takole:

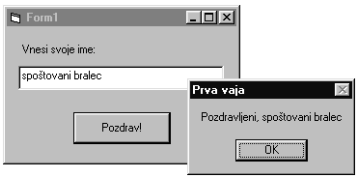
```
MsgBox "Pozdravljeni, " & Text1.Text
```

Celotna programska koda pa takole:

```
Private Sub Command1_Click()
  MsgBox "Pozdravljeni, " & Text1.Text
End Sub

Private Sub Form_Load()
  Text1.Text = ""
End Sub
```

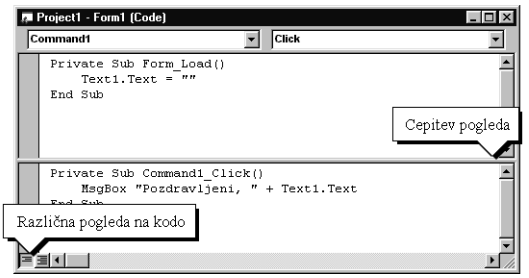
Preverite, ali je programska koda skladna z zgoraj izpisano, in poženite zgled.

Slika 3-18: Izpopolnjeni zgled



NAMIG Okno za urejanje kode ponuja še dva uporabna pripomočka, ki sta označena tudi na sliki 3-19.

Z gumboma pri dnu okna lahko preklapljamo s podprogramskega pogleda na celovit pogled na kodo (*procedure / full module view*) in nazaj. Podprogramski pogled nam hkrati prikazuje le en podprogram, med posameznimi podprogrami pa se pomikamo z uporabo obeh seznamov pri vrhu okna ali tipkama PgUp in PgDn. Večinoma se izkaže, da je bolje uporabljati celovit pogled, ki je izbran na začetku.



Slika 3-19: Pomočnika pri pregledovanju programske kode

Drugo uporabno orodje je cepitev pogleda. Ima svoj gumb pri vrhu navpičnega drsnika, ki ga lahko z miško zgrabimo in odvlečemo navzdol. Z njim lahko okno s kodo razcepimo na dva neodvisna dela, kar nam pride zelo prav, kadar želimo uporabiti kako vrstico iz že napisane kode kje drugje. Razcepljenega pogleda se znebimo tako, da gumb cepitve odvlečemo nazaj do vrha.

Če smo zgled že shranili, opazimo, da je naslov okna, ki ga prikaže podprogram MsgBox, enak imenu projekta (»Prva vaja«).

NAMIG Ker je zgled precej posodobljen, ga morda želimo shraniti pod novim imenom. Vendar moramo biti pazljivi. Kadar posodobimo obstoječi, že shranjeni projekt in bi ga radi shranili pod novim imenom, moramo *najprej* ročno shraniti vsak posamezen obrazec. Če bi projekt zgolj shranili pod novim imenom, bi se samodejno shranili tudi vsi obrazci, a pod že znanim imenom, in tako prekrili staro različico. Vsak obrazec izberemo v projektnem oknu in ga shranimo z ukazom **Save Form As...** iz glavnega menuja **File**. Na koncu izberemo še ukaz **Save Project As...** Ukazi za shranjevanje so na voljo tudi iz priročnega menuja projektnega okna. Vse skupaj je precej nerodno, če bi radi pripravili več generičnih projektov, ki jih delno posodobljene večkrat uporabimo. Zato so v visual basicu na voljo predloge (*templates*), ki jih bomo spoznali v nadaljevanju.

Pokukajmo naprej

Še ena končna izboljšava: ko program poženemo, ne moremo kar neposredno vnesti imena. Najprej moramo z miško ali s tipko Tab izbrati vnosno polje, šele nato lahko vnesemo ime. Program bi bil mnogo prijaznejši, če bi omogočal kar takojšen vnos imena.

Na voljo imamo *metodo* SetFocus, ki postori natančno to, kar želimo: vnosno točko postavi v izbrani element. Tega pa ne moremo postoriti kar v podprogramu

`Form_Load`, ker bo to povzročilo napako. Metoda SetFocus zahteva, da je obrazec že prikazan. Zato v oknu s kodo izberemo nov podprogram: `Form_Activate`. Z levega seznama izberemo element **Form**, z desnega pa podprogram **Activate**. Med izbiranjem podprograma opazimo, da so že uporabljeni podprogrami na desnem seznamu označeni s krepko pisavo.

Celotna programska koda je sedaj takale (novi del je označen krepko):

```
Private Sub Command1_Click()
  MsgBox "Pozdravljeni, " & Text1.Text
End Sub

Private Sub Form_Activate()
  Text1.SetFocus
End Sub

Private Sub Form_Load()
  Text1.Text = ""
End Sub
```

Če poženemo tako popravljen zgled, smo morda zadovoljni s tem, da nam je že takoj omogočen vnos imena, moti pa nas še nekaj. Pozdrav bi se lahko prikazal, brž ko vnesemo ime in pritisnemo tipko Enter. Sedaj tipka Enter zgolj povzroči nadležen pisk.

Rešitev je zelo preprosta. Gumbu zgolj nastavimo lastnost Default na True (angl. za *res*). Mimogrede, poizkusite preklapljati lastnost Default in hkrati opazujte videz gumba. Opazili boste, da je gumb, katerega lastnost Default je resnična vrednost, obrobljen bolj debelo.

> **NAMIG** Lastnosti, ki so lahko nastavljene le na omejen nabor možnosti, določamo s seznamom, ki se prikaže v tabeli lastnosti. Eno od njih zelo hitro izberemo, če seznam dvakrat kliknemo z miško.

Če zdaj poženemo zgled, končno deluje, kakor si želimo. Uganili bi lahko, da lastnost Default, nastavljena na True, določa, da se gumb sproži tudi po pritisku na tipko Enter. Če imamo na vmesniku več gumbov, je edino smiselno, da je tak le eden. Ponavadi je to tisti, ki potrdi vnos.

No, morda smo s temi izboljšavami malo prehitevali. Saj sploh še nisem povedal, kaj je to metoda, kaj dogodek in kaj vse je mogoče postoriti z navadnim gumbom! Res, a samo zato, da bi čim prej prišli do delujočega programa, s katerim bi se lahko naprej zabavali. Da pa bi bili resnično kos kar največ nalogam, ki so pred nami, si moramo stvari sistematično ogledati. Temu je posvečeno naslednje poglavje.

4. POGLAVJE

Osnove razvoja

Verjetno bo za vse zanimivo spoznati temeljna načela vizualnega programiranja na način visual basica, ki seveda temeljijo na skupnih značilnostih vseh programov, ki se izvajajo v Oknih. Gradniki in odzivni podprogrami uporabnika visual basica precej varujejo pred umazanimi podrobnostmi, pa vendar ne more škodovati, če jih vsaj bežno poznamo.

Okna na hitro

Družina operacijskih sistemov Microsoft Windows (Okna) ima *grafični* uporabniški vmesnik (GUI, *graphical user interface*). Za tovrstne vmesnike so značilna prekrivajoča se okna, menuji, ikone ter izdatna uporaba miške ali podobne kazalne naprave. Taka so Okna navzven, za vsakdanje uporabnike. Mi pa smo le nekaj več, mi Okna krotimo s programi, pisanimi v visual basicu. Zato moramo pojem »okno« nekoliko razširiti.

Za programerja je *okno* (*window*) pravokotno območje na zaslonu določenega videza in vedenja. Osnovno okno programa s svojo naslovno vrstico je okno, okno je tudi vsak gumb, vsako vnosno polje, celo menu. Vsa ta okna so vzorno označena in evidentirana v srcu operacijskega sistema. Operacijski sistem jih redno preverja in *obvešča* o najrazličnejših *dogodkih* (*events*), ki jih lahko zanimajo. Za gumb je tak dogodek prav gotovo klik z miško nad njegovo površino. Dogodkom, ki jih povzroči miška ali tipkovnica, pravimo tudi uporabniško ustvarjeni dogodki (*user events*). To pa še zdaleč niso vsi dogodki, o katerih je okno lahko obveščeno. Dogodek lahko povzroči tudi drugo okno ali pa izvira iz operacijskega sistema, npr. »končajte delo, pospravljamo«.

Če se zavedamo, da je oken na značilnem uporabniškem vmesniku toliko in da vsakega lahko doleti silno veliko dogodkov, je jasno, da bi popolno obvladovanje vse te množice sporočil (*messages*) terjalo zelo veliko dela. Visual basic zna večino stvari postoriti namesto nas. Po drugi strani pa nam omogoča, da določene dogodke obravnavamo po lastnih željah, tako kakor v našem prvem zgledu v prejšnjem poglavju. Zato lahko zelo hitro pridemo do delujočega programa, hkrati pa razvijemo program, izpiljen v vseh podrobnostih.

Dogodkovno programiranje

Druga značilnost operacijskega sistema Okna je večopravilnost (*multitasking*). Hkrati lahko uporabljamo več programov. Medtem ko se prenaša naša elektronska pošta, lahko urejevalnik besedil tiska novoletne voščilnice, mi pa polagamo pasjanso. Tudi večopravilnost omogoča sistem poizvedovanja in obveščanja o dogodkih, ki ga v celoti nadzira operacijski sistem. Zato pravimo, da so programi *dogodkovno vodeni*.

Dogodkovna naravnanost pomeni, da program nima vnaprej določene poti izvajanja. Vsak program, ki kakorkoli komunicira z uporabnikom, je pri današnji hitrosti računalnikov večinoma brezposeln – čaka, da premaknemo miško ali pritisnemo na tipko. V tem času lahko koristno delo opravljajo drugi programi, npr. tisti, ki skrbi za tiskanje ali prenašanje podatkov. Brž ko se uporabnik odzove, operacijski sistem to prestreže, sestavi ustrezno sporočilo ter z njim obvesti vse čakajoče stranke. Naš program pa ne more vedeti, ali bo uporabnik najprej pritisnil na tipko ali bo prej z miško kaj prestavil ali bo celo njegov prvi odziv na naš umotvor negativen in bo želel program čimprej zapreti. Za povrh smo že omenili, da ni samo uporabnik tisti, ki povzroča dogodke.

Pri razvijanju dogodkovno vodenih programov je nujno, da se zavedamo te nedoločenosti poti izvajanja. Zato moramo sestavne dele programa razvijati tako, da se izvajajo samo ob ustrezno izpolnjenih pogojih. V prvi vaji bi lahko preprečili izpis pozdrava, dokler bi uporabnik ne vnesel svojega imena. To bi bilo mogoče precej preprosto doseči, tako da bi onemogočili gumb, dokler bi uporabnik česa ne vnesel v vnosno polje.

Ker visual basic ustrezno obdela vse življenjsko pomembne dogodke, lahko brez težav poganjamo tudi na pol dokončane programe. To je zelo pogosto pri programiranju, saj bi drugače zlahka pozabili na marsikatero malenkost, ki se vidno odrazi šele po praktičnem preizkusu programa.

Gradnik, lastnost, metoda in dogodek

Čas je, da si sistematično ogledamo osnovne elemente, ki jih srečujemo pri izgradnji programa v visual basicu. Z večino smo se že praktično srečali v zgledu iz prejšnjega poglavja, nekatere smo celo bežno opisali. V nadaljevanju jih bomo natančno spoznali.

Gradnik

Gradnike (*controls*) uporabljamo pri načrtovanju uporabniškega vmesnika. Zbrani so v orodjarni (*Toolbox*). Po potrebi lahko dodajamo nove, ki jih lahko razvijemo tudi v samem visual basicu. Njihov videz določajo lastnosti (*properties*), vedenje pa odzivni

podprogrami (*event procedures*). Temeljni gradnik uporabniškega vmesnika je *obrazec* (*form*), gradniki pa so tudi vsi elementi, ki jih naknadno narišemo nanj, kot sta npr. gumb ali vnosno polje.

Lastnost

Dokončen videz posameznega gradnika določajo njegove *lastnosti* (*properties*). Najpomembnejša lastnost prav vsakega gradnika je njegovo *ime* (lastnost Name). Ime enolično določa sleherni gradnik na obrazcu in omogoča njegovo uporabo iz programske kode. Lastnosti gradnika določajo tudi njegov položaj in velikost na obrazcu ter kup drugih značilnosti, ki pa se od gradnika do gradnika precej razlikujejo, zato jih bomo spoznali malo pozneje, ko bomo gradnike obravnavali posamič. Nabor lastnosti je za vsak gradnik že določen, hkrati pa ima vsaka lastnost že svojo privzeto nastavitev, tako da v praksi spreminjamo nastavitve le manjšega dela lastnosti.

Dogodek in odzivni podprogram

Odzivni podprogrami določajo vedenje gradnika. Sprožijo se *samodejno* ob določenih *dogodkih* (*events*). Vrste odzivnih podprogramov, in s tem tudi prepoznani dogodki, so za vsako posamezno vrsto gradnika že dokončno izbrani. Programer se lahko zgolj odloča, katere odzivne podprograme bo uporabil in katerih ne bo, torej na katere dogodke se bo odzval in na katere ne. Odziv na dogodek zapišemo s programsko kodo v visual basicu. Njegovo skladenjsko zgradbo in uporabo bomo še natančno spoznali.

Kakor smo že omenili, je dogodek lahko posledica dejavnosti uporabnika (premik miške, pritisk na tipko ...), lahko je sporočilo drugega programa ali celo operacijskega sistema. Programer v visual basicu pa se mora zavedati, da je dogodek lahko tudi posledica ukaza v programski kodi.

Metoda

Metoda (*method*) je sestavni del gradnika, podobno kakor lastnost. Metoda je neko zaokroženo opravilo, ki ga zna gradnik opraviti. Dejansko je podprogram, vezan na posamezno vrsto gradnika. Podobno kakor vsi gradniki poznajo lastnosti Left in Top, ki določata njun položaj, poznajo tudi metodo Move, ki omogoča premik gradnika drugam. Ni redko, da metode opravljajo delo, ki ga je sicer mogoče postoriti tudi z nastavitvijo nekaj lastnosti. Seveda so tudi opravila, ki jih z lastnostmi ni mogoče doseči, zato moramo, da bi gradnik lahko učinkovito uporabljali, poznati tako njegove lastnosti kakor metode, pa tudi dogodke, ki jih prepozna.

NAMIG V razvojnem okolju visual basica je na voljo izčrpna pomoč v elektronski obliki. Če vas kak gradnik podrobneje zanima, ga postavite na obrazec in izberite ter pritisnite na tipko F1, ki prikliče sprotno pomoč. Ta prijem lahko uporabite tudi pri vnašanju poljubne lastnosti v okno Properties.

5. POGLAVJE

Uporaba gradnikov

Zdaj že vemo, da v visual basicu ne bomo mogli razvijati učinkovitih programov, dokler se natančno ne spoznamo vsaj z osnovnimi gradniki. Gradniki so pač sestavni del vsakega programa in se jim ne moremo izogniti. Hkrati nam izdatno olajšajo programiranje, saj za večino drobnih stvari poskrbijo sami, mi pa se lahko osredotočimo na srčiko problema.

V nadaljevanju bomo spoznali nekaj ključnih gradnikov, ki jih uporabljamo pri načrtovanju uporabniškega vmesnika, skupaj z osnovnimi napotki o njihovi rabi. Dogodki, metode in lastnosti bodo razloženi za vsak gradnik posebej, vendar že opisanih lastnosti, ki imajo enak pomen, ne bomo ponavljali. Pravih praktičnih zgledov uporabe tu še ne bo, saj si še nismo pobliže ogledali programiranja v visual basicu. Temu je namenjeno poglavje, ki sledi. Kljub temu bomo zapisali nekaj vrstic programske kode, ki se bodo ponovile v zgledih praktičnega programiranja pozneje. To poglavje vsekakor morate prebrati, če želite razumeti nadaljnje zglede.

Obrazec

Obrazec (*Form*) je temelj uporabniškega vmesnika. Je predstavnik *vsebnikov* (*containers*). To so gradniki, ki lahko vsebujejo nove gradnike. Vsako okno, ki ga bo uporabnik videl med izvajanjem programa, načrtamo znotraj obrazca, imamo pa tudi obrazce, ki ostanejo ves čas izvajanja skriti. Oglejmo si njegove *značilne*[4] lastnosti:

Lastnosti

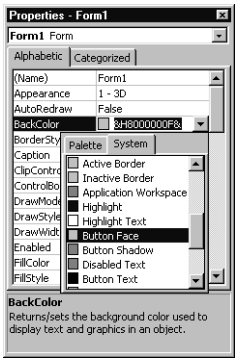
Lastnost **BackColor**

To je barva površine gradnika. Poleg obrazca pozna to lastnost še večina drugih gradnikov, velika izjema je le gumb (CommandButton), katerega površina se vedno

[4] Naš namen ni, da bi podrobno spoznavali popolnoma vse lastnosti. V ta namen je na voljo izčrpna pomoč, ki jo lahko uporabljamo v razvojnem okolju. Spoznali bomo tipične lastnosti, ki bodo bolje pojasnile sam pomen pojma »lastnost« pri programiranju v visual basicu. Podobno velja tudi za metode in dogodke.

sklada s sistemskimi nastavitvami, čeprav tudi pozna barvo ozadja[5]. Pri vseh nastavitvah barve nam visual basic ponudi posebno okno, v katerem so na eni strani zbrane sistemske barve, na drugi pa lahko izbiramo poljubno barvo s palete. Če želimo, da se barve našega programa ujemajo (in spreminjajo) s sistemskimi nastavitvami (barvna shema, *color scheme*), izbiramo raje barve s strani, ki se skriva pod jezičkom System.

Slika 5-1: Priporočljivo je, da barve izbiramo s sistemske palete.



Lastnost **BorderStyle**

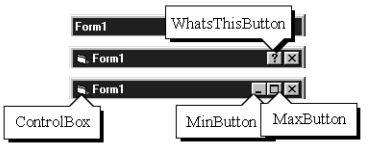
To je obroba gradnika. Pri obrazcu je ta lastnost še posebej pomembna. Določa namreč videz in posredno tudi namen okna, ki ga bo obrazec predstavljal med izvajanjem programa. Izbiramo lahko med šestimi različnimi obrobami, ki so označene s celimi števili od 0 do 5. Tovrstno označevanje bomo opazili precej pogosto, zato si velja zapomniti, da lahko v programski kodi uporabljamo izključno *številke*, v oknu za nastavljanje lastnosti pa je vsaka številka pojasnjena še z opisnim imenom. Pri obrazcu je lastnost BorderStyle mogoče nastaviti le med načrtovanjem programa (*design time property*), med izvajanjem programa pa je ne moremo več spreminjati, lahko pa po njej poizvedujemo.

Obroba obrazca neposredno vpliva še na nekaj lastnosti, ki so zbrane v naslednji tabeli. Vse naštete lastnosti je mogoče nastaviti zgolj na resnično (*true*) ali neresnično (*false*) vrednost:

[5] Barva ozadja se lahko odrazi tudi pri ukaznem gumbu, če ta ni povsem pravokotne oblike. Uporabniki starejših, 16-bitnih različic visual basica vemo, da so bili tam gumbi na vogalih zaokroženi, zato so bila oglišča obarvana v barvi ozadja. Če se bo Microsoft kdaj v prihodnosti odločil spremeniti obliko gumbov, lahko nastavitve barve ozadja zopet postanejo vidne. Zato je verjetno najbolje, da se te nastavitve pri gumbih ne dotikamo. Izjema je grafična oblika gumba, ki jo bomo spoznali v nadaljevanju.

Lastnost	Opis	Vpliv obrobe
ControlBox	Določa vidnost ikone nadzornega menuja in posredno tudi drugih gumbov.	Pri obrobi 0 nima vpliva.
MaxButton	Določa omogočenost gumba za maksimiranje okna.	Pri obrobi 0, 3, 4 in 5 nima vpliva.
MinButton	Določa omogočenost gumba za minimiranje okna.	Pri obrobi 0, 3, 4 in 5 nima vpliva.
ShowInTaskbar	Določa, ali je okno vidno tudi v opravilni vrstici.	Samodejno onemogočeno pri obrobah 4 in 5.
WhatsThisButton	Določa omogočenost gumba za opisno pomoč.	Na voljo samo pri obrobah 1, 3 in pogojno pri 2 (če sta MinButton in MaxButton neresnični).

Slika 5-2: Gumbi na obrobi okna



Lastnost **Caption**

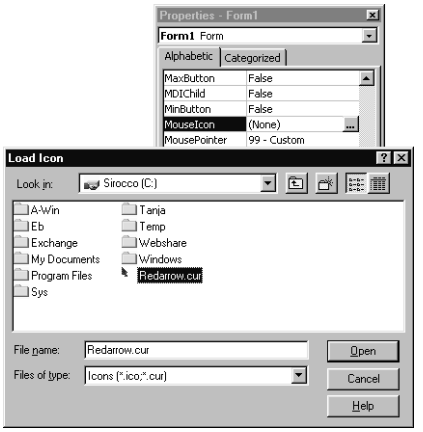
Lastnost Caption določa napis na gradniku, ki ga uporabnik ne more spremeniti. Najdemo jo pri obrazcu, pa tudi pri oznaki, stikalih in še kje, zaman pa jo bomo iskali npr. pri vnosnem polju, saj tam na napis – bolje rečeno, *vsebino* – neposredno vpliva prav uporabnik programa. Ko je obroba izključena (lastnost BorderStyle je nastavljena na 0), napis na oknu, ki ga določa lastnost Caption, ni viden, se pa kljub temu lahko odrazi v opravilni vrstici (*taskbar*) ali pod ikono pomanjšanega okna.

Lastnost **MousePointer**

Lastnost določa obliko kazalca miške nad gradnikom. Izbiramo lahko med 16 pripravljenimi oblikami ali pa izberemo oznako 99 (*custom cursor*). V tem primeru moramo določiti tudi lastnost MouseIcon, ki kaže na datoteko z definiranim kazalcem miške. Ta datoteka je lahko v standardnem zapisu kazalca (.CUR) ali ikone (.ICO)[6].

[6] Priprava tovrstnih datotek je mogoča s posebnimi orodji. Nekoč je bil urejevalnik ikon tudi sestavni del primerov programiranja v Visual Basicu. Pri novejših različicah Visual Basica je na njegovem CD-ROMu na voljo posebno orodje, ki ga najdemo v mapi \TOOLS\IMAGEDIT\. Zmogljiv urejevalnik ikon je na voljo tudi v orodju Microsoft Developer Studio, nekateri pa še vedno prisegamo na Borlandovo orodje Resource Workshop, ki so ga prilagali svojim starejšim razvojnim okoljem (npr. Borland C++ 4). Morda ga je še mogoče kupiti tudi v samostojni obliki?

NAMIG Nekaj tovrstnih datotek je priloženih visual basicu. Poglejte v mapo GRAPHICS\CURSORS\, ki jo najdete tam, kamor ste namestili (popoln) visual basic.



Slika 5-3: Za uporabo lastnih oblik kazalca miške potrebujemo zunanjo datoteko z obliko kazalca.

Lastnost **Moveable**

Določa, ali je okno med izvajanjem programa mogoče premikati (nastavitev *true*) ali ne (*false*).

Lastnost **StartUpPosition**

Te lastnosti smo se že bežno dotaknili pri obravnavanju pomožnega okna FormLayout, ki omogoča določitev začetne postavitve okna. To lahko neposredno nastavimo tudi s to lastnostjo, pri čemer imamo na izbiro štiri možnosti, od ročne nastavitve (0), kot jo določata lastnosti Left in Top, prek sredinskega položaja (2) do privzetega položaja (3), ki ga namesto nas izbere operacijski sistem. Nastavitev 1 (*center owner*) je uporabna pri vmesnikih s podrejenimi okni, ki jih bomo spoznali malo pozneje.

Lastnost **WindowState**

S to lastnostjo lahko določimo ali poizvedujemo po trenutnem stanju okna. Na voljo imamo običajno, minimirano ali maksimirano stanje, ki ga označujejo kode od 0 do 2.

Lastnost **Picture**

Da bi bili uporabniški vmesniki privlačnejši, lahko ozadje obrazca podložimo s sliko. Določa jo nastavitev lastnosti Picture, ki ji podamo ime datoteke z bitno ali vektorsko sliko (prepozna najrazličnejše oblike zapisa, tudi GIF in JPEG). Koristno je uporabiti vektorske slike, ki se vedno razširijo čez vso razpoložljivo površino

obrazca, pri bitnih slikah pa se izplača določiti nespremenljivo velikost obrazca (z lastnostjo BorderStyle).

NAMIG Mnogo uporabnih slik dobimo skupaj z visual basicom.

Lastnost **Enabled**

Ta lastnost določa, ali je gradnik omogočen (*enabled*). Z drugimi besedami, onemogočen gradnik se *ne* odziva na uporabniško povzročene dogodke (še vedno ga lahko nadziramo s programsko kodo). Pri celotnem obrazcu jo redko nastavljamo na neresnično vrednost (*false*), saj to pogosto ni smiselno. Drugače je pri posameznih gradnikih – večina jih to lastnost pozna –, kjer marsikaterega začasno onemogočimo.

Lastnost **Visible**

Ta lastnost določa vidnost gradnika. Če je nastavljena na neresnično vrednost (*false*), je gradnik neviden. Pri obrazcih jo le redko nastavljamo, saj za prikaz in skrivanje obrazcev uporabljamo metodi Show in Hide (spoznali ju bomo v nadaljevanju). Povsem drugače je pri drugih gradnikih, ki jih pogosto začasno skrivamo prav s to lastnostjo.

Lastnost **KeyPreview**

S to lastnostjo določimo posredovanje dogodkov, ki ponazarjajo stanje tipkovnice (KeyDown, KeyUp, KeyPress). Če je nastavljena na resnično vrednost (*true*), se dogodek najprej posreduje obrazcu, šele zatem gradniku, ki ima vnosno točko. Več o vnosni točki boste izvedeli pri gradniku oznaka, več o dogodkih tipkovnice pa pri gradniku vnosno polje.

Metode

Obrazec je gradnik s popolno podporo grafiki. Večina metod, ki jih podpira, je zato vezanih na izpisovanje besedila in izrisovanje grafike. Teh se zaenkrat ne bomo dotikali, saj jih bomo natančneje spoznali, ko bomo obravnavali delo z grafiko. Tu si oglejmo *značilne* metode obrazca, ki so uporabne predvsem v programih z več obrazci.

Metoda **Show**

S to metodo prikažemo skriti obrazec. Vsak program v visual basicu ima lahko več obrazcev, izmed katerih je največ eden prikazan samodejno. Vse druge moramo prikazati eksplicitno, s klicem metode Show.

Metoda **Hide**

Nasprotno metodi Show ta metoda skrije prikazani obrazec. Omenili smo že, da je učinek nekaterih metod mogoče doseči z nastavitvijo nekaj lastnosti. Metodi Hide

odgovarja nastavitev lastnosti obrazca Visible na neresnično vrednost (*false*). Te analogije pa ni mogoče uporabiti tudi za metodo Show, saj ta postori mnogo več, kakor zgolj spremeni vidnost obrazca.

Metoda **SetFocus**

Metodo smo že spoznali, ko smo izboljševali naš prvi zgled. Uporabna je tudi za obrazce, predvsem pa pride prav, ko imamo hkrati prikazanih več oken. Metoda SetFocus prestavi *vnosno točko* (*focus*) na izbrano okno. Več o pomenu vnosne točke sledi pri obravnavanju oznak, vnosnih polj in gumbov.

Metoda **PrintForm**

Pri vseh obrazcih je na voljo zelo priročna metoda, ki natisne njihovo podobo na dejavni tiskalnik. Ne moremo je sicer uporabiti za vse izpise, saj izpiše popolno podobo okna, skupaj z morebitno obrobo, toda nemalokrat pride prav.

Dogodki

Ključ do uporabne programske kode, ki določa vedenje našega programa, je poznavanje dogodkov, ki se lahko pripetijo določenemu gradniku. Obrazec kot temeljni gradnik programa ima še posebej pomemben nabor dogodkov, ki jih moramo natančneje spoznati.

Dogodek **Load**

Dogodek Load je prvi dogodek, ki se pripeti pri uporabi obrazca, pri katerem lahko iz kode že posegamo do gradnikov na njem. Če imamo projekt s samo enim obrazcem, se bo ta dogodek pripetil samo ob začetku programa in nikoli več. Zato je primerno mesto za nastavljanje lastnosti, ki se bodo odrazile na pozneje prikazanem obrazcu.

Če ne verjamete, da obrazec ob dogodku Load še ni viden, lahko preizkusite zgled:

```
Private Sub Form_Load()
  MsgBox "Obrazec še ni viden!"
End Sub
```

Dogodek **Initialize**

Dogodek Initialize je podoben dogodku Load, vendar so ga v visual basic dodali pozneje. Na splošno se dogodek Initialize pokliče za vse predmete (*objects*) ob njihovi vzpostavitvi (*instantiation*). Pri obrazcu lahko brez težav še naprej uporabljamo dogodek Load, pri drugih predmetih, ki jih bomo še spoznali, pa je drugače. Dogodek Initialize se pripeti pred Load, vendar v njegovem odzivnem podprogramu še ne moremo uporabljati gradnikov.

Dogodka **Activate** in **Deactivate**

Dogodek Activate se zgodi, kadarkoli se obrazec prikaže in je dejaven, to pomeni, da ima vnosno točko (*focus*). Če imamo v projektu samo en obrazec, se bo dogodek Activate pripetil samo ob prvem prikazu okna. Povsem drugače je, če imamo v projektu več obrazcev. Tedaj se dogodek Activate pripeti vedno, kadarkoli se vnosna točka prestavi v to okno (npr. če ga kliknemo z miško).

Praktično uporabo odzivnega podprograma za dogodek Activate smo spoznali, ko smo izboljševali prvi zgled uporabe visual basica.

Dogodek Deactivate bomo zasledili samo pri projektih z več obrazci. Pripeti se obrazcu, ki je izgubil vnosno točko v korist drugega obrazca. Vsak dogodek Deactivate tako spremlja dogodek Activate v nekem drugem oknu.

Dogodek **Resize**

Dogodek Resize se pripeti ob spremembi velikosti okna, vendar prav gotovo vsaj enkrat tudi ob prvem prikazu okna. Z lastnostjo BorderStyle lahko onemogočimo spreminjanje velikosti okna in s tem naknadno proženje tega dogodka.

Dogodek **Paint**

Dogodek Paint se pripeti vedno, kadar je postalo okno znova vidno in ga je treba osvežiti. V Oknih se lahko hkrati izvaja več programov, od katerih ima lahko vsak več oken. Vsa ta okna se med seboj prekrivajo in vsako mora biti kadarkoli sposobno obnoviti svojo površino. Visual basic nas reši večine skrbi, saj za obravnavo dogodka Paint poskrbi sam. Nekaj dela bomo imeli le z uporabo lastne grafike, vendar o tem pozneje.

> **NAMIG** Vrstni red dogodkov od zagonu projekta z enim samim oknom je zato tak: Initialize/Load/Resize/Activate/Paint. Pri tem se Initialize in Load izvedeta pri še nevidnem obrazcu.

Dogodek **DblClick**

Dogodek se pripeti, če površino obrazca, ki je ne prikriva noben drug gradnik, z miško dvakrat kliknemo.

Dogodka **QueryUnload** in **Unload**

Tudi zapiranje okna spremljajo posebni dogodki. Najprej si oglejmo par dogodkov QueryUnload in Unload. Dogodka nam omogočata, da pred zaprtjem okna še kaj postorimo, oziroma zapiranje celo prekličemo.

Glava odzivnega podprograma za dogodek QueryUnload je naslednja:

```
Private Sub Form_QueryUnload(Cancel As Integer, UnloadMode As Integer)
```

Za razliko od dogodkov, obravnavanih doslej, QueryUnload spremlja še dodatno pojasnilo v obliki *parametrov* Cancel in UnloadMode. Parameter Cancel pravzaprav ni dodatno pojasnilo, ampak rabi za morebiten preklic zapiranja okna. Pravo dodatno pojasnilo je parameter UnloadMode, ki opisuje vzrok zapiranja okna. Lahko ima naslednje vrednosti:

- Vrednost 0 – uporabnik je izbral ukaz Close iz nadzornega menuja okna.
- Vrednost 1 – obrazec želimo zapreti programsko s *stavkom* Unload.
- Vrednost 2 – operacijski sistem končuje sejo (odjava uporabnika – *log off* ali priprava za izklop – *shutdown*).
- Vrednost 3 – operacijski sistem (upravnik opravil, *Task Manager*) želi zapreti naš program.
- Vrednost 4 – zapira se nadrejeno okno (ta vrednost se prikaže samo pri vmesnikih MDI).

Če želimo zapiranje okna preprečiti, postavimo posredovani parameter na vrednost, različno od nič. Spremembe parametra so vidne tudi zunaj podprograma, zato vplivajo na nadaljnji potek izvajanja[7].

Podprogram dogodka QueryUnload najpogosteje uporabimo zato, da uporabnika opozorimo na zapiranje programa in mu omogočimo shranjevanje odprtih, spremenjenih dokumentov. Tak zgled bi bil lahko videti takole:

```
Private Sub Form_QueryUnload(Cancel As Integer, UnloadMode As Integer)
  Cancel = MsgBox("Zaprem okno?", 33, "Opozorilo") - 1
End Sub
```

Tu smo vgrajeni podprogram MsgBox prvič uporabili tudi kot *funkcijo*. O funkcijah več v naslednjem poglavju, uporabo podprograma MsgBox pa bomo natančno pojasnili ob reševanju praktičnih problemov.

Dogodek Unload prav tako spremlja parameter Cancel, vendar ničesar ne pove o vzroku zapiranja okna:

```
Private Sub Form_Unload(Cancel As Integer)
```

Kakor vidimo, je tudi tu mogoče preklicati zapiranje, vendar je med dogodkoma pomembna razlika. Če je bil vzrok za zapiranje okna eden izmed sistemskih dogodkov (zgoraj opisani paramer UnloadMode je imel vrednost 2 ali 3), bo preklic zapiranja okna v dogodku QueryUnload preprečil tudi nadaljnje izvajanje tega dogodka, česar se z odzivom na dogodek Unload ne da več preprečiti.

[7] Bolj izkušeni programerji vedo, da je to zaradi tega, ker se parameter prenese v podprogram po *referenci*. Način prenašanja parametrov bomo spoznali pri obravnavi podprogramov.

NAMIG Uporaba parametra Cancel seveda ni vsemogočna. Če želi kdo drug nasilno preprečiti izvajanje našega programa (z upravnikom opravil, Task Managerjem), mu tega ne moremo preprečiti[8].

Dogodek **Terminate**

Zadnji dogodek, ki ga bomo omenili pri obrazcih, je dogodek Terminate. Ravno nasproten je dogodku Initialize, saj je zadnji dogodek, ki se pripeti obrazcu. Tudi tu ni več mogoče posegati po gradnikih na obrazcu, zato se njegovi rabi navadno izogibamo, saj imamo na voljo dogodek Unload. Tudi Terminate na splošno uporabljamo pri predmetih, vendar ob njihovem uničenju.

Oznaka

Oznaka (*Label*) je gradnik, ki rabi za izpisovanje pojasnil in obvestil. Uporabnik programa na besedilo oznake ne more neposredno vplivati. Nastavljamo ga lahko izključno z lastnostjo Caption, torej ali med oblikovanjem ali pa s programsko kodo. Z oznako lahko opremimo gradnike, ki sicer nimajo lastnosti Caption. Z oznako najpogosteje pospremimo vnosno polje (TextBox).

Lastnosti

Lastnosti **Enabled** in **Visible**

Tudi ti dve lastnosti smo že spoznali pri obravnavanju obrazca, zato se tu le spomnimo, kaj počneta. Lastnost Enabled določa, ali je gradnik omogočen, lastnost Visible pa, ali je viden med izvajanjem programa.

Lastnost **Caption**

O lastnosti Caption smo povedali že marsikaj, ne pa vsega. Zamolčali smo, denimo, da z njo lahko določimo *dostopno tipko* (*access key*). To je kombinacija tipke Alt in še enega znaka na tipkovnici, ki je na zaslonu prikazan podčrtano. Cel kup dostopnih tipk najdemo npr. v menujih. Dostopno tipko določimo tako, da želeni znak označimo s predznakom & (*ampersand*). Če npr. želimo, da je v oznaki s pojasnilom »Vnesi ime« dostopna tipka kombinacija Alt+V, določimo lastnost Caption:

```
Label2.Caption = "&Vnesi ime:"
```

Zadeva deluje le, če je hkrati nastavljena tudi lastnost UseMnemonic, in sicer na resnično vrednost (*true*), kar je privzeta nastavitev. Če želimo izpisati prav znak &,

[8] Upravnik opravil (*Task Manager*) v Oknih ponavadi da programu nekaj časa, da konča svoje izvajanje. Če se v tem času izvajanje ne konča, ponudi nasilno prekinitev (*end task*), ki je programsko ne moremo preprečiti.

lahko izklopimo to lastnost in s tem izgubimo možnost dostopne tipke, ali pa znak zapišemo dvakrat zapored: &&.

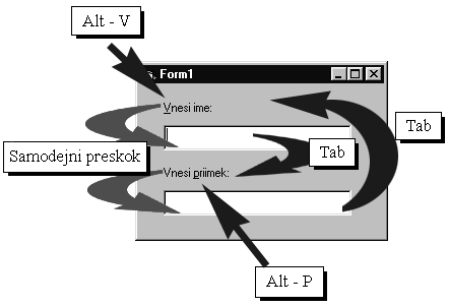
Ker oznaka ne prepozna tipkovnice, ne more prejeti vnosne točke (*focus*). Zato dostopna tipka ne sproži oznake, ampak gradnik, ki ji sledi v *dostopnem zaporedju* (*tab order*). Več o njem pri naslednji lastnosti.

Lastnost **TabIndex**

V Oknih lahko v določenem trenutku samo eno okno dobiva sporočila tipkovnice ali miške. Za to okno pravimo, da si lasti *vnosno točko* (*focus*). Pri delu z miško se omejitev vnosne točke niti ne zavedamo, saj je narava dela z njo taka, da samodejno sproži tudi gradnik, s katerim se želimo ukvarjati. Povsem drugače je s tipkovnico. Tega se prav dobro zavedamo, če poskušamo izpolniti obrazec z mnogo vnosnimi polji. Najprej moramo sprožiti ustrezno vnosno polje, šele zatem lahko vanj kaj vnesemo.

Vrstni red ali *dostopno zaporedje*, po katerem se sprožijo vnosna polja, pa tudi drugi elementi, ki lahko sprejmejo vnosno točko, je določen z lastnostjo TabIndex, ki jo ima praktično vsak gradnik. Lastnost je nenegativno celo število (od 0 dalje), ki se samodejno dodeli vsakemu gradniku, ki ga narišemo na obrazec. Po želji lahko vrstni red spremenimo. Kateremukoli gradniku lahko lastnost TabIndex spremenimo med načrtovanjem ali pa celo med izvajanjem programa. Ob vsaki spremembi visual basic samodejno popravi vse druge indekse, tako da so vedno enolično določeni in brez vrzeli.

Med izvajanjem uporabljamo tipko Tab oziroma kombinacijo Shift+Tab, da se pomikamo po dostopnem zaporedju naprej oziroma nazaj, od gradnika do gradnika. Posebna gradnika, ki tudi poznata lastnost TabIndex, sta oznaka in okvir (*Frame*), ki ga bomo spoznali pozneje. Čeprav ne moreta prejeti vnosne točke, lahko pojasnjujeta pomen kakega drugega gradnika. Zato je pomembno, da si ta dva gradnika neposredno sledita v dostopnem zaporedju. Če pri načrtovanju točno vemo, kaj želimo, in rišemo vmesnik npr. v zaporedju oznaka, vnosno polje, oznaka, vnosno polje ... bo že privzeto nastavljeno dostopno zaporedje ustrezno. Drugače je, če naknadno dodamo ali odstranimo kak gradnik. Tedaj moramo ročno preveriti in po potrebi popraviti lastnosti TabIndex.



Slika 5-4: Preskok vnosne točke z rabo dostopnih tipk ali tipke Tab. Vrstni red določa dostopno zaporedje.

Kot že vemo, lahko vnosno točko tudi programsko spremenimo z rabo metode SetFocus. Na vnosno točko vplivajo še lastnosti Visible, Enabled in TabStop. Če je katerakoli izmed njih nastavljena na neresnično vrednost, gradnik ne more prejeti vnosne točke. Gradnik oznaka (*Label*) lastnosti TabStop nima, saj pravzaprav ne prejme vnosne točke (ta preskoči na naslednji gradnik v dostopnem zaporedju).

Lastnost **Font**

Z lastnostjo Font lahko nastavimo obliko znakov, ki jih gradnik uporablja pri izpisovanju. Lastnost pozna vsak gradnik, ki omogoča izpisovanje besedila.

Lastnost **Alignment**

Lastnost Alignment določa vodoravno poravnavo besedila znotraj površine, ki jo določimo oznaki. Izbiramo lahko med levo, desno in sredinsko poravnavo (označene po vrsti z 0, 1 in 2).

Lastnost **BackStyle**

Besedilo, izpisano v oznaki, lahko leži na barvni podlagi, kot jo določa lastnost BackColor, lahko pa je ozadje prosojno. To določi lastnost BackStyle, nastavljena na prosojnost (vrednost 1, *transparent*).

Lastnost **AutoSize**

Če lastnost AutoSize nastavimo na resnično vrednost (*true*), se bo površina oznake samodejno prilagodila besedilu. Če velikost oznake pozneje ročno popravljamo, je treba vnovič nastaviti tudi lastnost AutoSize.

Lastnost **WordWrap**

Besedilo oznake se lahko izpiše tudi v več vrstic. V tem primeru moramo površino, ki pripada oznaki, zožiti po širini, poglobiti in vklopiti lastnost WordWrap (na resnično vrednost, *true*).

Lastnost **ToolTipText**

V zadnjem času so postali zelo priljubljeni namigi (*tooltips*), ki se prikažejo ob kazalcu miške, če nad določenim elementom nekaj časa obmirujemo. Če želimo to zmožnost uporabiti tudi za naše gradnike, jim preprosto nastavimo lastnost ToolTipText.

Metode

Metoda **Move**

Metodo Move smo že omenili. Pozna jo skorajda vsak gradnik, omogoča pa hkratno spremembo položaja in velikosti gradnika. Če bi želeli enak učinek doseči z lastnostmi, bi morali v nekaterih primerih spremeniti kar štiri, kar bi se lahko opazilo. Npr. če želimo gradnik premakniti levo navzgor, bi morali spremeniti dve

lastnosti, tako da bi gradnik najprej skočil levo, potem pa navzgor (ali nasprotno, odvisno od tega, katero lastnost bi najprej popravili). Z metodo Move se gradnik premakne takoj. Zgled rabe:

```
Label1.Move 10, 500, 3000, 100
```

Stori enako kakor nastavitev štirih lastnosti:

```
Label1.Left = 10
Label1.Top = 500
Label1.Width = 3000
Label1.Height = 100
```

Metoda **Refresh**

Tudi metodo Refresh si deli kar nekaj gradnikov. Njena naloga je, da popolnoma izriše vsebino gradnika. Ponavadi je ne kličemo ročno, saj za obnavljanje vsebine gradnikov poskrbi sam visual basic. Uporabimo jo takrat, ko obnavljanje ni dovolj hitro. Vsekakor moramo zgled najprej preizkusiti in se naknadno odločiti za uporabo te metode.

Metoda **ZOrder**

Z metodo ZOrder lahko med izvajanjem dosežemo to, kar nam pri načrtovanju omogočata ukaza **Bring To Front** (klic metode ZOrder s parametrom 0 ali brez) in **Send To Back** (klic s parametrom 1) iz glavnega menuja **Format ➪ Order**.

Dogodki

Dogodek **Change**

Edini dogodek oznake, ki ga bomo omenili na tem mestu, je dogodek Change. Pripeti se samodejno, kadarkoli se med izvajanjem spremeni vsebina oznake, torej lastnost Caption. Pri uporabi tega dogodka lahko naletimo na precejšnje težave.

V čem je težava? Omenili smo že, da ne povzroči vseh dogodkov uporabnik. Nekatere sprožijo drugi programi, druge sam operacijski sistem, nekatere dogodke pa povzročimo tudi s programsko kodo. Če s programsko kodo spremenimo lastnost Caption, samodejno sprožimo tudi dogodek Change. Še enkrat: dogodek se pripeti samodejno, kadarkoli se med izvajanjem spremeni vsebina oznake, torej lastnost Caption. In kaj če to lastnost znova spremenimo prav znotraj odzivnega podprograma dogodka Change? Ha, znova povzročimo dogodek, podprogram pokliče samega sebe in tako naprej, v začaranem krogu. Povzročimo neskončno zanko in program se ne odziva več, saj je vseskozi zaposlen sam s seboj. Njegovo izvajanje lahko prekinemo le še nasilno.

> **NAMIG** Izvajanje programa v razvojnem okolju nasilno prekinemo s pritiskom na kombinacijo tipk Ctrl+Break.

A ni vedno tako hudo. Dogodek Change se namreč ne zgodi, če je nova vsebina gradnika enaka stari. Če torej gradniku priredimo lastnost Caption na nespremenljivo – *konstantno* – vrednost, se izognemo neskončni zanki.

Vnosno polje

Vnosno polje (*TextBox*) je osnovni gradnik za vnašanje podatkov. Čeprav se kaže kot enostavno enovrstično polje za vnašanje besedila, je v resnici pravi mali urejevalnik besedila, funkcionalno praktično enakovreden Beležki[9] (*Notepad*).

Lastnosti

Lastnost **Text**

Osnovna lastnost vnosnega polja je seveda Text. Podaja besedilo, ki se izpiše na površini gradnika. Podobno kakor večino drugih lastnosti lahko tudi lastnost Text nastavimo ali pa jo preberemo. Za razliko od lastnosti Caption lahko nanjo neposredno vpliva tudi uporabnik programa.

Lastnost **MaxLength**

S to lastnostjo lahko podamo največje število znakov, ki jih gradnik dopušča pri vnašanju. Privzeta nastavitev je 0, kar pomeni brez omejitev.

Lastnost **PasswordChar**

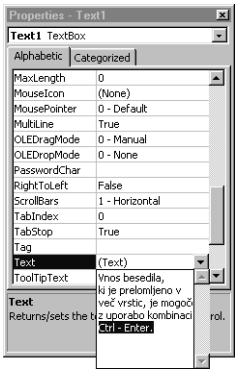
Včasih želimo vnašanje prikriti radovednim očem, npr. ko vnašamo geslo ali kako drugo občutljivo informacijo. Tu nam pomaga lastnost PasswordChar. Določa znak, ki vizualno predstavlja dolžino vnosa. Seveda lastnost Text še vedno odraža dejanski vnos.

Lastnost **MultiLine**

Ko lastnost MultiLine nastavimo na resnično vrednost (*true*), se enovrstično vnosno polje spremeni v zmogljiv urejevalnik besedila. Smiselno je, da v tem primeru povečamo površino, ki mu pripada. V večvrstični obliki pride do izraza tudi lastnost Alignment (glej razlago pri oznaki), ki se sicer zanemari.

> **NAMIG** Ko je vključena večvrstična različica, se spremeni tudi način nastavljanja lastnosti Text. Ne moremo ga več vnesti v pripadajočo vrstico, ampak moramo odpreti seznam, na katerem se prikaže večvrstični vnos. Če želimo vnesti lom vrstice, ne moremo pritisniti tipke Enter, saj ta konča vnašanje. Uporabiti moramo kombinacijo Ctrl+Enter.

[9] Preprost urejevalnik besedil, pomožni program, ki je sestavni del operacijskega sistema Windows.



Slika 5-5: Vnos lastnosti Text pri večvrstični različici vnosnega polja

Lastnost **ScrollBars**

Večvrstično vnosno polje samodejno premika besedilo, če ga je preveč za dano površino gradnika. Po besedilu se lahko premikamo s smernimi tipkami. Vseeno je praktično, da pri večvrstičnih vnosih omogočimo enostavnejše pomikanje besedila z *drsniki* (*scrollbars*). Tu nam pomaga ustrezna lastnost, ki drsnike bodisi izklaplja (nastavitev 0) bodisi vklopi vodoravnega (1), navpičnega (2) ali oba (3).

Nastavitev te lastnosti – če je vključeno levo poravnavanje – posredno vpliva tudi na prelom besedila.

Metode

Metoda **SetFocus**

Kar zadeva vnosno polje, je za nas ta hip zanimiva edino metoda SetFocus, katere pomen in učinek poznamo že vse od prvega zgleda. Poleg tega da prestavi vnosno točko v izbrani gradnik, zanj samodejno sproži tudi dogodek GotFocus

Dogodki

Dogodek **GotFocus**

Ko gradnik prejme vnosno točko, se samodejno sproži dogodek GotFocus.

Dogodek **LostFocus**

In ko vnosno točko izgubi, to pospremi dogodek LostFocus.

Dogodek **KeyPress**

Vsi gradniki, ki lahko prejmejo vnosno točko, poznajo dogodek KeyPress, ki se samodejno zgodi vsakič, ko pritisnemo na tipko – seveda, če je vnosna točka v opazovanem gradniku.

Dogodek KeyPress je za vnosno polje nadvse uporaben. Njegov čar je v tem, da se sproži, *še preden* se učinek tipke odrazi v vsebini vnosnega polja. To pomeni, da lahko uporabniku dobesedno gledamo pod prste in, če želimo, spreminjamo njegov vnos (nekaj podobnega smo počeli že pri dogodku QueryUnload, ki se nanaša na obrazec).

Oglejmo si glavo odzivnega podprograma dogodka KeyPress:

```
Private Sub Text1_KeyPress(KeyAscii As Integer)
```

Dogodek KeyPress spremlja še dodatno *pojasnilo*, vsebovano v *parametru* KeyAscii. To pojasnilo ni nič drugega kakor številska oznaka (koda ASCII[10]) znaka, ki ga uporabnik želi vnesti v vnosno polje. Najlepše pa je to, da se vse spremembe, ki jih napravimo na tem parametru v odzivnem podprogramu, odrazijo tudi navzven[11].

Če vas zanimajo kode ASCII, opremite odzivni podprogram z naslednjima vrsticama kode (označeni sta krepko):

```
Private Sub Text1_KeyPress(KeyAscii As Integer)
  Text1.Text = KeyAscii
  KeyAscii = 0
End Sub
```

Kaj smo storili? Namesto znaka smo v vnosno polje izpisali njegovo kodo, vneseni znak pa preprosto zanemarili. Vnosno polje sedaj tudi ne piska več, če v njem pritisnemo tipko Enter. Namesto tega ugotovimo, da tista tipka predstavlja »znak« s kodo 13. Ker tega »znaka« ni mogoče prikazati, je običajni odziv vnosnega polja – pisk. Vidimo tudi, da nezaželene znake zanemarimo tako, da parameter KeyAscii postavimo na nič.

Dogodek KeyPress pozna tudi sam obrazec. Če je njegova lastnost KeyPreview nastavljena na resnično vrednost, se pri vsaki tipki, ki jo pritisnemo, najprej pokliče odzivni podprogram obrazca, čeprav je vnosna točka kje drugje (npr. znotraj vnosnega polja). Tako lahko posebne tipke obravnavamo na enem mestu za vse gradnike na obrazcu.

Gumbi in stikala z okviri

Drug način komunikacije z uporabnikom je uporaba gumbov (*CommandButton*) in stikal (*CheckBox* in *OptionButton*). Uporabimo jih predvsem za enostavno vnašanje, oziroma za proženje določenih operacij. Ker en sam gradnik ponavadi ni preveč

[10] ASCII (*American Standard Code for Information Interchange*) je standardiziran način predstavitve števk, črk, ločil in nekaterih posebnih oznak v obliki številskih kod. Natančno določa kode 128 znakov, za kar je potrebno 7 dvojiških bitov.

[11] Bolj izkušeni programerji vedo, da je to zato, ker se parameter prenese v podprogram po *referenci*. Način prenašanja parametrov bomo spoznali pri obravnavi podprogramov.

uporaben, jih združujemo v skupine, ki jih včasih zaradi videza, včasih pa tudi zaradi funkcije, združujemo v okvire (*Frame*). Zaradi sorodnega namena bomo vso to pisano druščino gradnikov spoznali skupaj.

Lastnosti

Skupna pomembna lastnost vseh tu obravnavanih gradnikov je Caption, ki smo jo že spoznali pri oznaki. Tudi tu veljajo zanjo enake značilnosti, predvsem si zapomnimo možnost določanja dostopne tipke. Za posamezne gradnike so pomembne še naslednje lastnosti:

Najprej si oglejmo posebni lastnosti gumba.

Lastnost **Default**

Pomen lastnosti Default smo že spoznali v izboljšanem prvem zgledu. Če je nastavljena na resnično vrednost (*true*), je gumb mogoče sprožiti s pritiskom na tipko Enter, tudi če nima vnosne točke (*focus*). Lastnost Default nastavimo samo enemu gumbu na obrazcu. To je gumb, ki *potrdi* vse morebitne vnose in ponavadi okno tudi zapre.

Lastnost **Cancel**

Lastnost Cancel je nekako nasprotna lastnosti Default. Če jo nastavimo na resnično vrednost, je gumb mogoče sprožiti s pritiskom na tipko Esc, tudi če nima vnosne točke. Tudi to lastnost nastavimo samo enemu gumbu, ki *prekliče* morebitne vnose in ponavadi okno zapre.

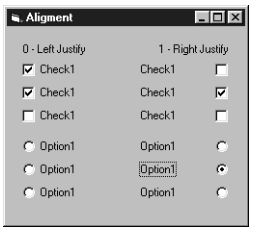
> **NAMIG** Medtem ko tipka Esc vedno izbere gumb z nastavljeno lastnostjo Cancel, je pri lastnosti Default treba nekaj previdnosti. Če je uporabnik prej s tipko Tab prenesel vnosno točko na kak drug gumb, bo tipka Enter sprožila izbrani gumb in ne tistega z nastavljeno lastnostjo Default. Gumb, ki ga sproži tipka Enter, prepoznamo po debelejši obrobi. Mimogrede, izbrani gumb ali stikalo sproži tudi pritisk na preslednico.

Stikala so dveh oblik. Navadno stikalo (*CheckBox*) omogoča vklop in izklop nastavitve. Prepoznamo ga po kvadratni obliki. Druga oblika stikala, opcijsko stikalo (*OptionButton*), je uporabna edino v skupinah. V skupini opcijskih stikal je lahko vključeno natančno eno. Uporabimo ga torej za vklop ene izmed omejenega števila nastavitev. To stikalo je okroglo. Posebne lastnosti stikal so:

Lastnost **Alignment**

Lastnost Alignment določa stran, na kateri je simbol stikala, glede na spremno besedilo, ki ga določa lastnost Caption. Simbol stikala je lahko na levi ali desni strani besedila:

Slika 5-6: Dve možnosti razvrstitve simbola stikala



Lastnost **Value**

Stanje, v katerem je stikalo, določa lastnost Value. Ta je za navadno stikalo (*CheckBox*) številska vrednost, ki določa:

- Vrednost 0 – izklopljeno stanje (*0 – Unchecked*).
- Vrednost 1 – vklopljeno stanje (*1 – Checked*).
- Vrednost 2 – nedoločno stanje (*2 – Grayed*).

Uporabnik lahko neposredno preklaplja le med prvima dvema vrednostma, medtem ko je stikalo mogoče postaviti v nedoločno stanje le med načrtovanjem ali s programsko kodo.

Za potrebe opcijskega stikala se lastnost Value prelevi v logično vrednost, ki je lahko resnična (*true*) ali neresnična (*false*), in s tem neposredno ponazarja stanje stikala. Že pri načrtovanju programa je mogoče lastnost Value nastaviti na resnično vrednost le za eno opcijsko stikalo. To se odrazi tudi med izvajanjem. Če uporabnik vključi kako opcijsko stikalo, s tem izključi prej izbranega. Edina izjema je stanje, v katerem so vsa stikala izključena.

Slika 5-7: Stanja stikal

Kako lahko na istem obrazcu določimo več med seboj neodvisnih skupin stikal? Na to vprašanje bo odgovoril gradnik okvir (*Frame*). Še prej pa si oglejmo zanimivo možnost popestritve vmesnika.

> **NAMIG** Lastnost Value je na voljo tudi za ukazni gumb. Če jo nastavimo na resnično vrednost (*true*), posredno sprožimo dogodek Click.

Lastnost **Style**

Gumb in stikala poznajo lastnost Style, ki je po privzeti nastavitvi v stanju 0 (navadna oblika, *0 – Standard*), po želji pa jo lahko preklopimo v stanje 1, s čimer

postanejo omenjeni gradniki *grafični* (*1 – Graphical*). Z grafičnimi gradniki lahko določimo poseben videz gumbov in stikal. Pri tem so nam v pomoč še nekatere lastnosti.

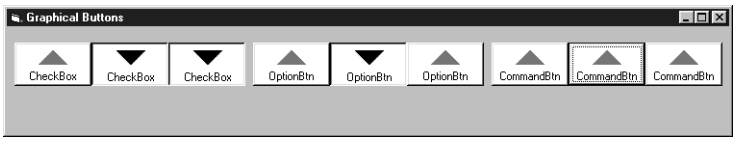
> **NAMIG** Če izberemo grafične gradnike, vsi postanejo nekakšni gumbi. Videz navadnih gumbov se ne spremeni bistveno, le da jim sedaj lahko dodajamo še grafiko, navadna stikala postanejo gumbi, ki so lahko neodvisno vklopljeni ali izklopljeni, bolje rečeno, ugreznjeni ali ne, opcijska stikala pa se prelevijo v skupine gumbov, v katerih je lahko vklopljen (ugreznjen) le eden.

Lastnosti **Picture**, **DownPicture** in **DisabledPicture**

Dokončen videz grafičnega gradnika določajo bitne slike, ki jih moramo prej pripraviti in shraniti v eni izmed oblik, ki jih visual basic prepozna[12]. Lastnost Picture določa sliko, ki se izriše na izklopljenem gumbu. Lastnost DownPicture določa sliko, ki se izriše na vklopljenem gumbu. Pri navadnem gumbu (*CommandButton*) je ta slika vidna le za hip, drugje pa ponazarja vklopljeno stanje.

Če želimo, lahko določimo tudi lastnost DisabledPicture. Ta slika se na gumbu izriše, ko je ta onemogočen (njegova lastnost Enabled je nastavljena na neresnično vrednost – *false*). Če te slike ne določimo, bo (ne)ustrezno sliko za onemogočeno stanje izdelal sam visual basic.

> **NAMIG** Če izberemo grafični slog gradnikov, je koristno nastaviti še lastnost BackColor. Ta v tem primeru deluje tudi pri navadnih gumbih (*CommandButton*).



Slika 5-8: V grafični obliki postanejo tudi stikala gumbi, ki pa se še vedno vedejo kakor njihovi negrafični predstavniki.

Lastnosti **MaskColor** in **UseMaskColor**

Tudi pri grafičnem slogu je barva ozadja gumba določena z lastnostjo BackColor. Če za ozadje izberemo barvo s sistemske palete, se lahko zgodi, da bo uporabnik

[12] Kot smo že omenili pri lastnosti Picture za sam obrazec, je to večina priljubljenih zapisov bitnih slik, kot so standardni okenski zapisi CUR, ICO, DIB in BMP, standarda v svetovnem spletu GIF ter JPEG, lahko pa uporabimo tudi vektorski zapis WMF (*Windows Meta File*).

spremenil sistemske nastavitve in slike na gumbih bodo lahko postale preveč opazne, z vidnimi zunanjimi robovi. Zato je koristno, da lahko za barvo ozadja izberemo prosojno in se tej težavi izognemo. Pri tem nam pomagata lastnosti MaskColor (ta določa barvo na sliki, ki postane prosojna) in UseMaskColor (ki mora biti v tem primeru nastavljena na resnično vrednost (*true*)).

Najbolje je, da ozadje obarvamo v živo osnovno barvo, ki jo zlahka najdemo na paleti za določitev barv visual basica. Seveda ta barva ne sme biti uporabljena tudi na sliki. Najpogosteje je uporabna čista rdeča, zelena ali modra barva.

> **NAMIG** Še priročneje je izdelati sliko, ki že vsebuje prosojno barvo. Tak zapis je mogoč v obliki CUR (*Windows Cursor*), ICO (*Windows Icon*) ali GIF (*Graphic Interchange Format*, različica 89a). V tem primeru bo visual basic samodejno prepoznal prosojno barvo.

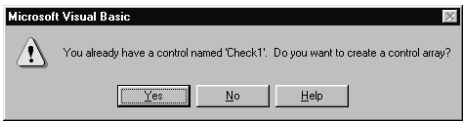
Grafične oblike gumbov so zanimiva popestritev vmesnika, najbolj pa so uporabne za izdelovanje orodjarn in vrstic z gumbi, podobnim tistim v visual basicu. Boljše različice visual basica imajo tudi posebne zunanje gradnike (v obliki ActiveX), ki dodatno poenostavijo izgradnjo orodjarn.

Lastnost **Index**

Stikala in gumbi na uporabniškem vmesniku so pogosto v skupinah. Še posebej opcijska stikala, ki posamič sploh niso uporabna. Koristni programi morajo obravnavati vsako stikalo in se ustrezno odzvati na njegov vklop oziroma izklop. Mar to pomeni, da bomo morali pri uporabi 10 stikal pisati 10 zelo podobnih odzivnih podprogramov? Na srečo ne.

Vsi gradniki poznajo poleg lastnosti Name tudi lastnost Index, s katero lahko rešimo omenjeno težavo. Omenili smo že, da lastnost Name enolično določi vsak gradnik na uporabniškem vmesniku in nam omogoča njegovo naslavljanje iz programske kode. Z lastnostjo Index lahko določimo *isto* ime več gradnikom *iste* vrste. Zahtevana enoličnost pa je zagotovljena z *različnimi* zaporednimi številkami, ki so zapisane v lastnosti Index.

Lastnosti Index lahko nastavimo ročno, gre pa tudi drugače. Pri načrtovanju vmesnika z mnogo stikali so si ta med seboj pogosto zelo podobna. Zato najprej popolnoma določimo le enega, potem pa ga z odložiščem (*clipboard*) razmnožimo v več izvodih. Izbrani gradnik lahko z ukazom **Copy** iz priročnega menuja (ali iz glavnega menuja **Edit**) prepišemo na odložišče in z ukazom **Paste** napravimo njegovo kopijo. Če se ne dotikamo lastnosti Index, nas bo pri poizkusu izdelave prve kopije visual basic povprašal (slika 5-9).

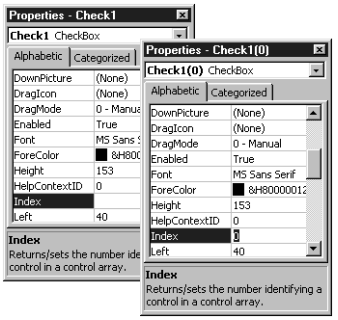


Slika 5-9: Opozorilo pred izdelavo polja gradnikov

S pritrdilnim odgovorom bomo ustvarili *polje gradnikov*. Polje gradnikov (*Control Array*) je skupina gradnikov iste vrste z enakim imenom. Visual basic zagotavlja, da ima vsak element polja gradnikov edinstveno zaporedno številko, ki je nenegativno celo število, določeno z lastnostjo Index.

> **NAMIG** Zaporedne številke gradnikov v polju niso nujno zaporedne. Brez težav imamo lahko polje štirih gradnikov z zaporednimi številkami npr. 0,1,10,223. Največje zaporedno število je 32.767. Zaporedno število lahko zato v nekaterih primerih uporabimo tudi kot nosilec koristne informacije.

Seveda lahko polje gradnikov ustvarimo tudi z ročnim nastavljanjem lastnosti Index. Če je lastnost Index prazna, gradnik ni element polja gradnikov in mora imeti zato enolično določeno ime (lastnost Name). Nastavitev lastnosti Index se takoj odrazi v oknu za nastavljanje lastnosti gradnika:



Slika 5-10: Ime gradnika z nastavljeno lastnostjo Index je drugačno, kar je razvidno s seznama pri vrhu okna.

Polje gradnikov se občutno odrazi na vseh odzivnih podprogramih. Oglejmo si, kako je sedaj videti glava odzivnega podprograma dogodka Click:

```
Private Sub Check1_Click(Index As Integer)

End Sub
```

Vsi odzivni podprogrami gradnika, ki je element polja gradnikov, sedaj prejmejo dodaten parameter Index, ki je enakovreden istoimenski lastnosti. Tako lahko znotraj enega samega odzivnega podprograma obdelamo isti dogodek za vse gradnike iz polja, dejavnega pa določimo s tem novim parametrom. Kako se spremenijo glave podprogramov, ki tudi sicer prejmejo parameter?

```
Private Sub Check1_KeyPress(Index As Integer, KeyAscii As Integer)
End Sub
```

Pričakovano. Tudi tu se doda nov parameter Index.

> **NAMIG** Velika slabost razvojnega okolja visual basica je enosmernost vizualnih orodij. To pomeni, da se spremembe na uporabniškem vmesniku, ki so ključne za delovanje programske kode, v njej ne odrazijo.
>
> Če za kak gradnik napišemo odzivno proceduro, zatem pa mu spremenimo ime ali pa ga uvrstimo v polje gradnikov, stara procedura ne deluje več. Ročno je treba napisati novo. Zato velja posvetiti vizualnemu načrtovanju vmesnika dovolj pozornosti, da se po začetku kodiranja ne ukvarjamo več s poimenovanjem gradnikov in ustvarjanjem novih polj gradnikov.

Lastnost **Tag**

V enem izmed namigov zgoraj smo omenili, da je zaporedno številko gradnika v polju mogoče izkoristiti kot nosilca dodatne (celoštevilčne) informacije. Dostikrat želimo tudi kakemu gradniku, ki ni element polja, dodati novo informacijo, ali pa nam številke ne zadoščajo. V obeh primerih nam pride prav lastnost Tag, ki jo pozna vsak gradnik in *nima* vnaprej določenega pomena.

Lastnosti lahko priredimo poljubno vrednost, kako jo bomo uporabili, pa je v celoti odvisno od programske kode, ki jo napišemo.

> **NAMIG** Lastnost Tag je bila v starejših različicah visual basica zgolj celo število (podobno kakor lastnost Index). V petici je lahko poljuben *podatkovni tip*.

Izmed vseh gradnikov, ki smo jih našteli v naslovu, nam je ostal le še okvir (Frame).

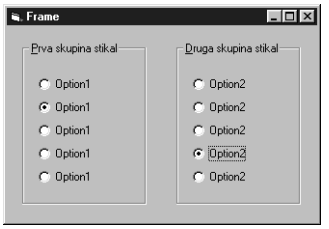
Lastnost **BorderStyle**

Okvir (*Frame*) uporabljamo za vizualno, a tudi funkcionalno združevanje sorodnih gradnikov. Okvir je, podobno kakor obrazec, tudi predstavnik vsebnikov (*containers*), torej gradnikov, ki lahko vsebujejo druge gradnike. Najuporabnejši je za ločevanje skupin opcijskih stikal. Kakor vemo, je v skupini teh stikal lahko vključeno natančno eno samo. Če želimo na isti obrazec postaviti več med seboj neodvisnih skupin opcijskih stikal, moramo poseči po okviru.

Gradnike uvrščamo v okvir tako, da jih izrišemo znotraj okvira. Če okvira še nismo narisali, nam pomaga naslednji recept:

- Označimo vse gradnike, ki jih želimo prestaviti v okvir.
- Iz priročnega menuja ali iz glavnega menuja **Edit** izberemo ukaz **Cut**.
- Narišemo okvir.
- Pri izbranem okviru (okoli površine, ki jo prekriva, so izrisane ročice) iz priročnega menuja ali menuja **Edit** izberemo ukaz **Paste**.

Ali je določen gradnik v okviru ali ne, najlaže preizkusimo, če okvir premaknemo. Če se skupaj z njim premakne tudi opazovani gradnik, je gotovo v okviru.



Slika 5-11: Neodvisni skupini opcijskih stikal je mogoče ustvariti z gradnikom okvir.

Če nas obroba okvira moti, a bi kljub temu radi imeli več neodvisnih skupin opcijskih stikal, lahko posežemo po nastavitvi lastnosti BorderStyle in jo nastavimo na 0 (*None*).

Metode

Opis in pomen metod **Move**, **Refresh**, **SetFocus** in **ZOrder**, ki jih pri gumbih, stikalih in okvirih največ uporabljamo, smo že spoznali (glej obravnavo gradnika *oznaka*).

Dogodki

Za stikala in gumbe je daleč najpomembnejši in največkrat uporabljeni dogodek vsekakor ...

Dogodek **Click**

Za navadni gumb smo ga že spoznali, uporabljamo pa ga tudi pri stikalih. Stikala največkrat določimo kot polje gradnikov in v odzivnem podprogramu dogodka Click spremenimo nastavitev glede na posredovano zaporedno številko gradnika (*index*):

```
Private nastavitev

Private Sub Option1_Click(Index As Integer)
  nastavitev = Index
End Sub
```

```
Private Sub Command1_Click()
  MsgBox "Izbrana nastavitev je " & nastavitev
End Sub
```

Seznami

Zelo pogosta komunikacija z uporabnikom je uporaba seznamov. Visual basic pozna dve osnovni vrsti seznamov: zaprtega (*ComboBox*) in odprtega (*ListBox*). Zaprti se naprej deli še v tri podvrste. Oglejmo si njihove značilnosti.

Lastnosti

Lastnost **List**

Lastnost List določa vsebino seznama. Vnašamo jo podobno kakor besedilo lastnosti Text pri večvrstični obliki. Vsaka vrstica je nov element seznama, nove vrstice pa dodajamo s pritiskom na kombinacijo tipk Ctrl+Enter.

Lastnost List se v programski kodi vede kakor *polje* (*array*). Polja bomo natančneje spoznali malo pozneje, zaenkrat omenimo le to, da moramo za dostop do posamezne vrstice seznama poznati tudi njeno zaporedno številko (*index*), ki jo zapišemo v oklepaju. Prvo vrstico seznama lahko v programski kodi spremenimo tako:

```
List1.List(0) = "Nova vsebina prve vrstice"
```

Da, zaporedna številka prve vrstice je 0! Zgled seveda deluje le, če seznam že vsebuje kakšno vrstico. Kako s programsko kodo dodamo novo vrstico, pa bomo pojasnili, ko bomo obravnavali metode.

Lastnost **ItemData**

Vsaki vrstici seznama, ki se navzven vedno odraža kot besedilo, lahko predpišemo tudi celoštevilčno vrednost. V tem primeru izpolnimo še vrednosti lastnosti ItemData. Po privzeti nastavitvi je ta lastnost nastavljena na 0 za vsako vrstico seznama.

Zgledov uporabe te lastnosti je cel kup kar v oknu za nastavljanje lastnosti, Properties. Kakor smo že omenili, je cela vrsta lastnosti kodiranih s celimi števili (npr. BorderStyle). Čeprav s seznama izbiramo opisne vrednosti (npr. *1 – Fixed Single*), je dejanska nastavitev vedno le celo število, kar moramo upoštevati v programski kodi. Pretvorbo iz opisne vrednosti v številsko najlaže dosežemo prav z rabo lastnosti ItemData.

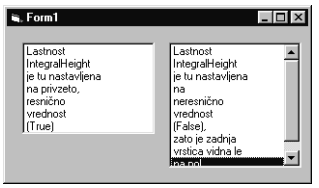
Lastnost **Sorted**

Lastnost določa, ali se elementi seznama samodejno uredijo po abecedi. Žal imamo pri rabi te lastnosti nekaj nevšečnosti. Nerodno je že to, da je izid urejanja viden šele po tem, ko program poženemo, še huje pa je, da se skupaj z opisnimi vrsticami ne preuredijo tudi ustrezne številčne vrednosti v lastnosti ItemData. Če se zavedamo teh pomanjkljivosti, bomo urejanje lahko s pridom uporabili, še posebej, ker upošteva tudi značilnosti slovenske abecede[13].

Lastnost **IntegralHeight**

Odprte sezname in posebno obliko zaprtega seznama uporabljamo v večvrstični obliki. Pri določanju velikosti gradnika pa največkrat uporabljamo mrežo (*grid*), zato višina gradnika ne more biti poljubna. Višina mreže na eni in vrstic seznama na drugi strani pa se največkrat ne ujemajo popolnoma, zato lahko izrišemo seznam, na katerem je nekaj vrstic povsem vidnih, zadnja pa le deloma. To težavo odpravi lastnost IntegralHeight, nastavljena na resnično vrednost (*true*), ki zagotavlja, da je višina gradnika celi večkratnik višine vrstice, ne glede na vklopljeno mrežo.

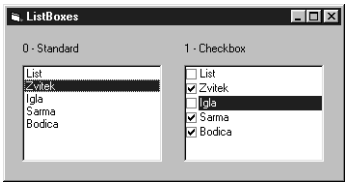
Slika 5-12: Vpliv nastavitve IntegralHeight



Lastnost **Style**

Ta lastnost določa podvrsto seznama. Pri odprtem seznamu imamo na voljo dve nastavitvi. Vidni sta na spodnji sliki:

Slika 5-13: Podvrsti odprtega seznama *ListBox*



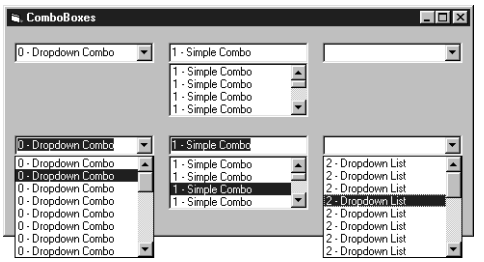
Običajna oblika odprtega seznama je zgolj nekaj vrstic besedila, posebna oblika, označena s kodo 1, pa vsako vrstico pospremi še s stikalom. Druga oblika je predvsem primerna namesto skupine med seboj neodvisnih stikal. Čeprav to na zgornji sliki ni očitno, je tudi pri običajni obliki odprtega seznama mogoče hkrati izbrati več vrstic. Kako to storimo, bomo spoznali pri nadaljnji obravnavi lastnosti.

---

[13] Pri tem predpostavljam, da uporabljate slovenska Okna ali vseevropsko različico (*paneuropean*) s slovenskimi krajevnimi nastavitvami (*regional settings*).

Spodnja slika prikazuje vse tri različne podvrste zaprtega seznama v običajni (zgoraj) in dejavni obliki (spodaj):

Slika 5-14: Podvrste zaprtega seznama *ComboBox*



Opraviti imamo s precej pisano druščino. Prava zaprta seznama sta samo dva (vrsti 0 in 2), medtem ko je t. i. Simple Combo pravzaprav odprti *kombinirani* seznam. Kombinirani seznam je tudi prvi (vrsta 0, Dropdown Combo), medtem ko je zadnji (vrsta 2, Dropdown List) pravzaprav zaprta oblika navadnega seznama. Kaj pomeni kombinirani seznam? *Kombinirani seznam* je kombinacija vnosnega polja in zaprtega seznama. Zato pozna vse značilnosti vnosnega polja, ki smo jih spoznali malo prej, in tudi vse značilnosti seznama, ki jih spoznavamo zdaj.

Lastnost **Text**

Pri uporabi zaprtega seznama (*ComboBox*) lahko trenutno izbrano vrednost vedno preberemo tudi iz lastnosti Text, saj smo spoznali, da je kombinirani seznam pravzaprav kombinacija vnosnega polja in seznama. Nastaviti pa jo je mogoče le pri pravih kombiniranih seznamih (lastnost Style je 0 ali 2), pri zaprtem seznamu Dropdown List pa je lastnost Text na voljo le za branje.

To ima tudi precej pomembno posledico: medtem ko je lastnost Text pri prvih dveh vrstah zaprtega seznama lahko bodisi vnaprej določena vrstica bodisi vnos uporabnika, je pri zadnjem zaprtem seznamu lastnost Text lahko le prazna ali pa izbrana s seznama. Uporabniški vnos na zaprti seznam Dropdown List ni mogoč. Če skušamo vanj vnesti kak znak, nam kvečjemu pokaže vrstico, ki se začenja s tem znakom (če seveda sploh je kaka taka).

Namen različnih zaprtih seznamov je s tem določen. Prvi dve obliki izberemo tedaj, ko želimo uporabniku ponuditi nekaj značilnih vnosov, hkrati pa mu omogočimo tudi vnašanje novih vrednosti. Zadnjo obliko zaprtega oziroma odprti seznam pa uporabimo tedaj, ko lahko uporabnik izbere le eno od vnaprej določenih vrednosti. Značilen zgled je standardno pogovorno okno za izbiranje pisave (uporabljamo ga pri nastavitvah lastnosti Font). Medtem ko je oblika znakov lahko le ena od možnosti s seznama, imamo pri velikosti dve možnosti: lahko izberemo eno od značilnih ali pa vnesemo svojo (npr. 33)[14].

[14] Dejansko je pogovorno okno izvedeno s tremi kombiniranimi seznami Simple Combo in programska koda skrbi za zgoraj naštete omejitve.

Ali uporabimo odprti ali zaprti seznam, je predvsem stvar oblikovanja vmesnika. Kadar uporabnika ne želimo obremenjevati s preveč informacijami, oziroma nas tare prostorska stiska, izberemo zaprti seznam.

Lastnost Text je za branje na voljo tudi pri odprtem seznamu (*ListBox*). Odprti seznami pa nam ponujajo še druge načine ugotavljanja izbrane vrstice ali celo več izbranih vrstic.

Lastnosti **ListIndex** in **ListCount**

To sta lastnosti, ki nista dostopni že med načrtovanjem programa, ampak ju lahko uporabimo le v programski kodi. Lastnost ListIndex podaja trenutno izbrano vrstico s seznama, pri čemer je prva vrstica vedno označena z 0, vrednost –1 pa pomeni, da ni izbrana še nobena vrstica! Lastnost ListCount podaja skupno število vrstic na seznamu. Spodaj zapisani zgled programske kode deluje pri vseh oblikah seznamov in izpiše vsebino trenutno izbrane vrstice:

```
Private Sub List1_Click()
  MsgBox List1.List(List1.ListIndex)
End Sub
```

Kako pa bi uporabili še dodatno številčno informacijo, zapisano v lastnosti ItemData?

```
Private Sub List1_Click()
  With List1
    MsgBox .List(.ListIndex) & " : " & .ItemData(.ListIndex)
  End With
End Sub
```

Stavek `With` nam olajša pisanje, saj nam ni treba vseskozi ponavljati imena gradnika, ki mu naštete lastnosti pripadajo. Kakor vidimo, preberemo izbrano vrstico v odzivnem podprogramu dogodka Click, kar nam hkrati zagotavlja, da je vrstica gotovo izbrana. Na splošno je lahko lastnost ListIndex tudi –1, kar ponazarja, da ni izbrana nobena vrstica. Na vmesnik lahko dodamo gumb, ki izključi morebiti izbrano vrstico:

```
Private Sub Command1_Click()
  List1.ListIndex = -1
End Sub
```

Nastavljanje lastnosti ListIndex je še posebej pomembno za tretjo podvrsto zaprtega seznama (*Dropdown List*). Kakor je bilo razvidno že s slike pri obravnavi lastnosti Style, je tovrsten seznam na začetku prazen (nobena vrstica ni izbrana, lastnost ListIndex ima vrednost –1). Če želimo, da ta zaprti seznam po privzeti nastavitvi že prikazuje neko vrstico, moramo dodati vrstico kode v odzivni podprogram dogodka Load za obrazec, na katerem je seznam (včasih obravnavamo tudi dogodka Activate ali GotFocus):

```
Private Sub Form_Load()
  Combo1.ListIndex = 2
End Sub
```

V tem primeru bo zaprti seznam na začetku prikazal vsebino tretje vrstice (ne pozabimo, da pripada prvi zaporedna številka 0).

> **NAMIG** Nastavitev lastnosti ListIndex samodejno povzroči tudi dogodek Click.

Lastnosti **MultiSelect** in **Selected**

Odprti seznam omogoča, da hkrati izberemo več vrstic. To je bilo očitno pri njegovi podvrsti, ki jo določa lastnost Style, nastavljena na 1. Če uporabljamo navadni odprti seznam (lastnost Style pustimo na privzeti nastavitvi 0), lahko omogočimo hkratno izbiranje več vrstic z lastnostjo MultiSelect. Nastavimo jo lahko na naslednje kode:

- Vrednost 0 – navadno izbiranje (*0 - None*). Mogoč je izbor največ ene vrstice.
- Vrednost 1 – enostavno izbiranje (*1 - Simple*). Mogoč je izbor nobene, ene ali več vrstic. Vsaka vrstica se vede kakor navadno stikalo (prvi klik jo vključi, drugi pa izključi).
- Vrednost 2 – razširjeno izbiranje (*2 - Extended*). Mogoč je izbor nobene, ene ali več vrstic. Posamezno vrstico izberemo kakor običajno, več sosednjih vrstic pa tako, da z miško povlečemo čeznje ali pa izberemo prvo, tiščimo pritisnjeno tipko Shift in izberemo zadnjo. Posamezno vrstico izberemo tako, da jo kliknemo, medtem ko tiščimo pritisnjeno tipko Ctrl. Mogoča je tudi souporaba tipk Shift in Ctrl.

> **NAMIG** Večina namenske programske opreme uporablja razširjeno izbiranje, enostavnega pa skoraj nikoli. Enostavno izbiranje lahko nadomestimo z uporabo druge oblike odprtega seznama, ki jo določa lastnost Style. Če je lastnost Style nastavljena na 1, lastnosti MultiSelect ni mogoče spreminjati.

Kako lahko izločimo vse izbrane vrstice? Tu nam je v pomoč lastnost Selected. Naslednji zgled opozori na vse izbrane vrstice:

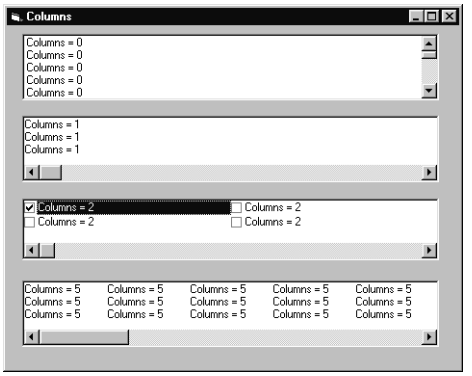
```
For i = 0 To List1.ListCount - 1
  If List1.Selected(i) Then
    MsgBox (i + 1) & ". vrstica je izbrana"
  End If
Next
```

Natančen opis *stavka* `If` in *zanke* `For` bomo spoznali, ko bomo obravnavali programsko kodo. Skupno število izbranih vrstic podaja lastnost SelCount.

Lastnost **Columns**

Odprti seznam navadno prikazuje vse vrstice, urejene v navpičnem stolpcu. Uredimo pa jih lahko tudi v enega ali več vodoravnih. To nam omogoča lastnost Columns. Njen učinek si lahko ogledate na spodnji sliki:

Slika 5-15: Učinek nastavitve lastnosti Columns



Vidimo, da je urejanje v več stolpcev mogoče pri obeh slogih odprtega seznama.

NAMIG Učinek nastavitve je le delno opazen med načrtovanjem. Žal moramo pri uporabi visual basica programe neredko pognati, da lahko preizkusimo želene nastavitve.

Metode

Vsebino seznama lahko v celoti določimo že med načrtovanjem. Za dinamično spreminjanje njegove vsebine med izvajanjem pa imamo na voljo nekaj metod:

Metode **Clear**, **AddItem** in **RemoveItem**

Če želimo odstraniti vse vrstice s seznama, uporabimo metodo Clear:

```
List1.Clear
```

Novo vrstico dodamo z metodo AddItem, ki jo lahko uporabimo v dveh oblikah:

```
List1.AddItem "Nova vrstica"
```

Doda novo vrstico na *konec* seznama. Po želji lahko metodi pošljemo še en parameter, ki določa vrstico, v katero se nova vsebina *vrine*:

```
List1.AddItem "Nova prva vrstica", 0
List1.AddItem "Nova tretja vrstica", 2
```

Na voljo imamo tudi lastnost NewIndex, ki vrne zaporedno številko nazadnje dodane vrstice.

NAMIG Lastnost NewIndex nam pomaga pri dodajanju vrstic na urejen seznam. Vedno jih dodajamo z metodo AddItem brez drugega parametra, saj je uvrstitev elementa odvisna od njegove vsebine. Z lastnostjo NewIndex lahko po dodajanju vnesemo še pripadajočo številčno vrednost v polje ItemData:

```
With List1
  .AddItem "Janez"
  .ItemData(.NewIndex) = 1000
  .AddItem "Poldi"
  .ItemData(.NewIndex) = 3200
  .AddItem "Anton"
  .ItemData(.NewIndex) = 5100
End With
```

Odstranjevanje vrstic je naloga metode RemoveItem. Metodi moramo posredovati zaporedno številko vrstice, ki jo želimo odstraniti. Če želimo odstraniti prvo vrstico, zapišemo:

```
List1.RemoveItem 0
```

Kako pa zadnjo?

```
List1.RemoveItem List1.ListCount - 1
```

Nikoli ne pozabimo, prvi vrstici pripada zaporedna številka 0! Kako že spremenimo vsebino obstoječe vrstice? Spomnimo se lastnosti List!

Dogodki

Namen in uporabo dogodka Click smo že spoznali, tudi pri praktičnih zgledih obravnave seznamov. Za zaprte sezname prvih dveh podvrst (Dropdown Combo in SimpleCombo) še enkrat omenimo, da vsebujeta tudi vnosno polje, zato lahko pri njiju smiselno uporabimo dogodka Change in KeyPress. Posebej omenimo le dogodek ItemCheck.

Dogodek **ItemCheck**

Odprti seznami posebne oblike (lastnost Style je 1), pri katerih vsako vrstico spremlja stikalo, poznajo dogodek ItemCheck. Njegov odzivni podprogram se samodejno pokliče ob preklopu kateregakoli stikala, kot parameter pa prejme zaporedno številko stikala, ki ustreza zaporedni številki vrstice, v kateri je.

Če smo zlobni, lahko onemogočimo vklapljanje stikala v 3. vrstici:

```
Private Sub List1_ItemCheck(Item As Integer)
  If Item = 2 Then List1.Selected(Item) = False
End Sub
```

Drsnika

Drsnika (*ScrollBar*) že poznamo kot sestavna dela nekaterih gradnikov (večvrstičnega vnosnega polja ali seznama), na voljo pa sta tudi kot samostojna gradnika. Uporabimo ju tedaj, ko želimo uporabniku omogočiti izbor vrednosti iz zveznega intervala. Neposredno se ta interval lahko razteza od –32.768 do 32.767, lahko pa ga s preračunavanjem tudi spremenimo.

Lastnosti

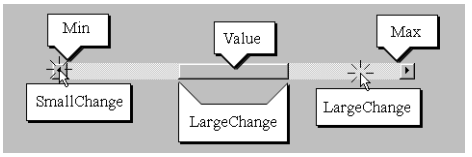
Lastnosti **Min** in **Max**

Z lastnostma Min in Max določimo želeni zvezni interval. Obe vrednosti morata biti celi števili v zgoraj omenjenih mejah.

Lastnosti **Value**, **SmallChange** in **LargeChange**

Trenutni položaj drsnika opisuje lastnost Value. Lastnosti SmallChange in LargeChange določata velikost spremembe trenutnega položaja pri uporabi drsnika. Ključne lastnosti in uporaba so delno razvidne s spodnje slike:

Slika 5-16: Lastnosti in uporaba drsnika



Pri manjši spremembi se trenutni položaj in s tem tudi vrednost Value spremeni za vrednost SmallChange, pri večji pa za vrednost LargeChange. Drsnik lahko prejme vnosno točko (*focus*), tako da ga je mogoče nadzorovati tudi s tipkami. Manjša sprememba se pripeti ob pritisku na smerni tipki, večja pa ob uporabi tipk PgUp in PgDn.

> **NAMIG** Ne glede na orientacijo drsnika, smerni tipki za levo in navzgor zmanjšujeta, za desno in navzdol pa povečujeta trenutno vrednost. Delujeta tudi tipki Home (drsnik preskoči na najmanjšo vrednost) in End (preskok na največjo vrednost).

Dogodki

Za uporabo drsnika sta najpomembnejša dva dogodka:

Dogodek **Change**

Dogodek Change se pripeti ob spremembi vrednosti Value, bodisi da jo povzroči uporabnik bodisi da je povzročena programsko, z nastavitvijo te lastnosti na novo vrednost.

Dogodek **Scroll**

Dogodek Scroll se dogaja *med* spreminjanjem vrednosti Value. Uporabnik ga lahko povzroči, če zgolj povleče gumb drsnika z miško. Če drsnik nadzorujemo le s tipkami, dogodka Scroll ne bo.

> **NAMIG** Dogodek Scroll se pripeti tudi gradnikom, ki vsebujejo drsnik, npr. večvrstičnemu vnosnemu polju ali seznamu.

Malo daljši zgled uporabe obeh omenjenih dogodkov:

```
Private Sub Form_Load()
  With HScroll1
    .Min = 0
    .Max = 255
    .SmallChange = 1
    .LargeChange = 10
    Form1.BackColor = RGB(.Value, 0, 0)
  End With
End Sub

Private Sub HScroll1_Change()
  With HScroll1
    Form1.BackColor = RGB(.Value, 0, 0)
    MsgBox "Nova barva: " & .Value * 100 \ .Max  & "% rdeče"
  End With
End Sub

Private Sub HScroll1_Scroll()
  Form1.BackColor = RGB(HScroll1.Value, 0, 0)
End Sub
```

Na začetku poskrbimo za nastavitev lastnosti, ki jih lahko opravimo tudi že med načrtovanjem vmesnika. V odzivnem podprogramu dogodka Change spremenimo barvo ozadja in obvestimo uporabnika o opravljeni spremembi. Ker se dogodek Change pokliče le ob koncu spreminjanja vrednosti, uporabnika ne bomo obremenjevali z vmesnimi sporočili. Koristno pa je, da se že med spreminjanjem vrednosti drsnika spreminja tudi barva obrazca. To postorimo v odzivnem podprogramu dogodka Scroll.

> **NAMIG** Ker si pri uporabi drsnika dogodki Scroll sledijo zelo hitro, se v njegovih odzivnih procedurah izogibamo dolgotrajnim opravilom ali komunikaciji z uporabnikom (npr. klicu podprograma MsgBox).

Ura

Ura (*Timer*) je edini vgrajeni gradnik, ki med izvajanjem programa nikoli ni viden. Zato je vseeno, kam ga umestimo med načrtovanjem vmesnika. Uporabimo ga za časovno usklajeno izvajanje določenih programskih stavkov.

> **NAMIG** V 32-bitnih različicah Oken, katerim je visual basic od različice 5 naprej tudi namenjen, skupno število ur praktično ni omejeno.

Lastnosti

Lastnost **Interval**

Lastnost Interval določa časovni interval, v katerem ura povzroča dogodek Timer. Čas podajamo v milisekundah. Čeprav vrednost intervala lahko nastavimo na krajše od 50 milisekund, je to spodnja meja zmožnosti gradnika (največ približno 18 dogodkov na sekundo). Prav tako ne moremo pričakovati prevelike natančnosti. Če interval nastavimo npr. natanko na eno sekundo, se lahko zaradi trenutne prezaposlenosti sistema dva zaporedna dogodka Timer sledita v daljšem, včasih pa celo v malo krajšem časovnem intervalu. Največja dovoljena nastavitev je 65.535, kar je nekaj več kakor minuto. Če je interval nastavljen na 0, je ura onemogočena.

Lastnost **Enabled**

Lastnost Enabled smo prvič omenili že, ko smo obravnavali obrazec. Tam smo zapisali, da lastnost določa, ali se gradnik odziva na uporabniško povzročene dogodke ali ne. Pri uri je malo drugače, saj je nevidna in zato uporabniško povzročenih dogodkov sploh ne pozna. Še vedno pa velja, da lastnost Enabled določa, ali je ura *omogočena* ali ne. Če je lastnost nastavljena na neresnično vrednost (*false*), se dogodek Timer ne bo sprožil.

Dogodki

Dogodek **Timer**

Edini dogodek, ki ga pozna ura, se imenuje Timer. Sproži ga ura sama, in sicer v podanem časovnem intervalu.

Za preprosto uro, ki prikazuje točen čas, potrebujemo vmesnik z oznako (*Label*) in uro, katere interval je nastavljen na 1000 milisekund. V odzivni podprogram zapišemo eno samo vrstico:

```
Private Sub Timer1_Timer()
  Label1.Caption = Now
End Sub
```

Menujska izbira

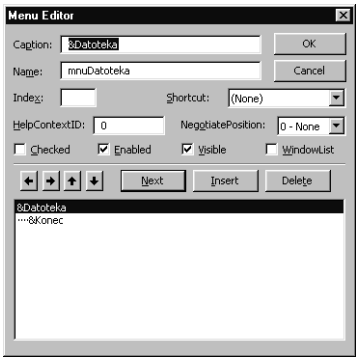
Tudi vsaka posamezna izbira v menuju je gradnik uporabniškega vmesnika. Žal nam razvojno okolje visual basica to značilnost menujskih izbir delno zakrije, saj jih oblikujemo v posebnem urejevalniku. Kljub temu so – s stališča programerja – menujske izbire še najbolj podobne gumbom in stikalom. Seveda pa imajo tudi nekaj posebnosti. O *izbiri* govorimo zato, ker je element menuja lahko ukaz, ločilo ali celo nov menu.

Načrtovanje menujev

Menujska vrstica vedno tiči ob vrhu okna, zato je ne rišemo, tako kakor druge gradnike, ampak menu načrtujemo v posebnem oknu, *Menu Editor*. Prikličemo ga lahko na več načinov:

- V priročnem menuju obrazca izberemo ukaz **Menu Editor**.
- Izberemo ukaz **Menu Editor** iz glavnega menuja **Tools**.
- Pritisnemo na kombinacijo tipk Ctrl+E.

Slika 5-17: Urejevalnik menujev



Urejevalnik menujev združuje vnosna polja in stikala za nastavljanje lastnosti (zgoraj) ter hierarhični urejevalnik menujskih izbir (spodaj, pod črto). V urejevalniku menujev moramo vsako izbiro opisati vsaj z njenim imenom (lastnost Name), tako kakor vsak drug gradnik. Ponavadi za vsako izbiro vnesemo še njen napis (lastnost Caption).

NAMIG V namenskih programih pogosto opazimo, da se napis na menujskih izbirah (pa tudi na ukaznih gumbih) končuje s tremi pikami (...). Tako označevanje je v navadi pri ukazih, ki odprejo pogovorno okno. Z drugimi besedami, ukaz, katerega ime se konča s tremi

pikami, se ne izvede takoj, ampak uporabnika še nekaj vpraša (prek pogovornega okna).

Zelo pomembno: tudi enostavnejši menuji ponujajo mnogo izbir. Ker je vsaka posamezna izbira gradnik, je zelo pomembno, da zanje izberemo smiselna imena (lastnost Name). To je še toliko pomembneje, ker nam visual basic ne ponudi privzetih imen, tako kakor za druge gradnike. Priporočam, da se vsako ime začne s predlogom **mnu**, ki mu sledi napis. Tako bi menujsko izbiro z napisom **Datoteka** poimenovali **mnuDatoteka**. Pazimo, da ne vnesemo presledka, ki v imenih ni dovoljen. Tako bodo menujske izbire na seznamih zbrane skupaj (ker so seznami abecedno urejeni), hkrati pa bomo že na prvi pogled razbrali *vrsto* gradnika.

NAMIG Pri obširnem naboru menujskih ukazov je dobro, če ime odraža tudi hierarhični položaj. Tako bi ukazu **Odpri** v menuju **Datoteka** lahko priredili simbolično ime **mnuDatotekaOdpri**.

Podobnega pravila se je dobro držati tudi pri poimenovanju drugih gradnikov, ki jim ročno popravimo lastnost Name. Ustrezne predloge bomo našteli pozneje, ko bomo obravnavali predmete.

Ker je vse lastnosti menujskih izbir mogoče nastavljati tudi v navadnem oknu (okno Properties), urejevalnik menujev izkoristimo predvsem za hierarhično ureditev izbir. Pri tem so nam v pomoč gumbi v spodnjem delu okna.

Po vnašanju posamezne izbire pritisnemo tipko Enter ali izberemo privzeti gumb **Next** (mimogrede se spomnimo lastnosti Default), ki nas prestavi v naslednjo vrstico. Če želimo, lahko na izbrano mesto na seznamu vrinemo novo izbiro (gumb **Insert**) ali pa že določeno izbiro odstranimo (gumb **Delete**). Določene izbire lahko preurejamo s štirimi gumbi s puščicami. Prva dva hierarhično zamikata izbire, druga dva pa omogočata spremembo položaja izbire.

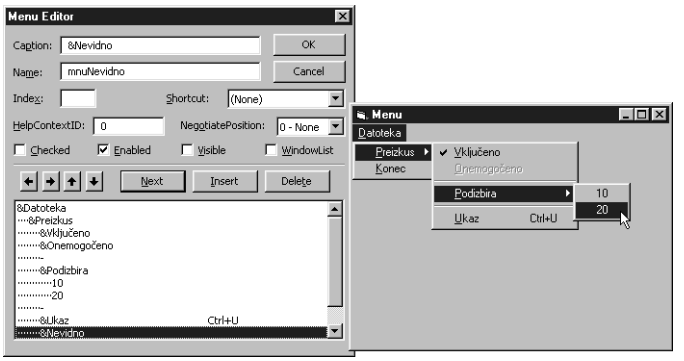
Vse vnesene izbire so na začetku na najvišjem nivoju. Izpisane so v menujski vrstici. Večinoma se pod njimi skriva seznam podrejenih izbir. Vse te izbire morajo biti za en nivo zamaknjene, kar dosežemo, če kliknemo gumb s puščico na desno. Po želji lahko tudi podrejenim izbiram dodamo nov podnivo in tako naprej. Visual basic dovoli štiri nivoje menujskih izbir.

Posebna oblika izbire je ločilo. Ločilo določimo tako, da za napis vnesemo en sam znak minus (»–«). Tudi ločila morajo biti pravilno poimenovana.

Na sliki 5-18 vidimo načrtovanje menuja in njegov videz med uporabo.

Kakor vidimo, je ena izbira odkljukana, ena onemogočena, ena pa celo nevidna. Vse to nadzirajo lastnosti, ki jih bomo spoznali spodaj.

Slika 5-18: Načrtani menu (levo) in njegov videz med delom (desno)

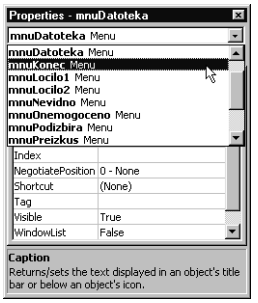


Tudi menujske izbire so lahko združene v polje gradnikov (*control array*), saj poznajo lastnost Index. Upoštevati moramo le eno omejitev: vsi elementi polja morajo biti na istem nivoju in podrejeni isti izbiri.

Lastnosti

Kakor rečeno, lahko lastnosti izbir že izdelanega menuja spreminjamo tudi z navadnim oknom za njihovo nastavljanje. Ker menuja ne moremo izbrati na samem obrazcu, moramo pred nastavljanjem lastnosti ustrezen gradnik poiskati na seznamu okna Properties. Tudi tu se izkaže, da se izplača vsako izbiro poimenovati s predlogom mnu ali podobnim, saj bodo tako zbrane skupaj.

Slika 5-19: Če izbire pravilno poimenujemo, jih je mnogo laže iskati.



Lastnost **Checked**

Lastnost Checked določa, ali je ustrezna menujska izbira odkljukana ali ne. Lastnosti ne moremo nastaviti za ločila in nadrejene izbire. Za razliko od stikal, ki sama skrbijo za svoje stanje, je pri menujski izbiri vse prepuščeno programerju. To je dobro, saj lahko skupino izbir obravnavamo v medsebojni odvisnosti (kot opcijska stikala) ali pa ne (kot navadna stikala). Zgled si bomo ogledali, ko bomo obravnavali dogodke.

Lastnost **Shortcut**

Lastnost Shortcut določa kombinacijo tipk (v računalniškem žargonu imenovano *bližnjica*), ki sproži menujsko izbiro. Tudi tu velja, da bližnjice ne moremo prirediti ločilu ali nadrejeni izbiri. Bližnjice so lahko le vnaprej določene kombinacije tipk (izbiramo jih s seznama) in so povečini kombinacija tipke Ctrl in nekega znaka.

Menujskim izbiram, tudi nadrejenim, lahko z lastnostjo Caption in znakom »&« (*ampersand*) določimo tudi dostopno tipko. Dostopne tipke smo natančneje spoznali, ko smo obravnavali lastnosti Caption za gradnik oznaka (*label*). V čem pa je razlika med dostopno tipko in bližnjico?

Kakor vemo, je dostopna tipka kombinacija tipke Alt in izpostavljenega znaka, ki je izpisan podčrtano. Da bi se uporabnik prebil do nekega ukaza znotraj menujske strukture, mora včasih pritisniti tudi več dostopnih kombinacij[15]. Bližnjica pa ukaz sproži takoj, ne glede na nivo, na katerem je določena izbira »zakopana«. To tudi pomeni, da mora uporabnik bližnjico poznati na pamet, dostopne kombinacije pa ne, saj zgolj sledi podčrtanim znakom.

Lastnost **WindowList**

To lastnost nastavimo na resnično vrednost (*true*) zgolj za izbiro na najvišjem nivoju pri vmesniku MDI. S tem se bo med podrejene izbire samodejno dodalo tudi ločilo in seznam vseh odprtih podrejenih oken, podobno kakor pri glavnem menuju Window v razvojnem okolju visual basica. Več o vmesnikih MDI bomo izvedeli, ko bomo obravnavali praktične probleme programiranja.

Dogodki

Dogodek **Click**

Omenili smo že, da so za programerja menujske izbire zelo podobne gumbom ali stikalom. To nakazuje tudi edini dogodek, ki ga poznajo, dogodek Click. V njegovem odzivnem podprogramu določimo funkcijo ukaznih izbir.

Spodnji zgled obravnava izbiro kot navadno stikalo:

```
Private Sub mnuPreklop_Click()
  mnuPreklop.Checked = Not mnuPreklop.Checked
End Sub
```

Metode

Menujske izbire ne poznajo posebnih metod, vendar se nanje navezuje metoda obrazca, ki je, ko smo jih obravnavali, nismo omenili. To je metoda **PopUpMenu**. Z njo lahko prikazujemo priročne menuje.

[15] V resnici je kombinacija s tipko Alt potrebna le pri odprtju menuja, pri nadaljnjem izbiranju zadošča, da pritiskamo zgolj tipko izpostavljenega znaka.

Metoda kot parameter prejme ime nadrejene izbire in prikaže priročni menu z vsemi pripadajočimi podizbirami. Pri načrtovanju menujev imamo torej dve možnosti:

- Priročni menu zgolj ponovi izbire, ki jih tudi sicer najdemo nekje v glavnem menuju.
- Pri načrtovanju menujev določimo posebno glavno izbiro, ki je nevidna (njena lastnost Visible je neresnična – *false*). Vse podizbire, ki ji pripadajo, tako v glavnem menuju ne bodo dosegljive, prikažemo pa jih lahko kot priročni menu. Pazimo! Zadošča, da je nevidna nadrejena izbira, izmed podrejenih pa mora biti vidna vsaj ena, drugače bo priročni menu prazen.

Če izberemo prvo možnost, bi lahko izkoristili že pripravljen menu iz zgornjega zgleda. Denimo, da želimo prikazati priročni menu v hipu, ko uporabnik klikne ukazni gumb:

```
Private Sub Command1_Click()
  Form1.PopupMenu mnuPreizkus
End Sub
```

Slika 5-20: Prikazan priročni meni

Seveda priročni menu najpogosteje prikažemo tako, da kliknemo z desnim gumbom miške nad določeno površino. Žal zaenkrat še ne znamo ugotoviti, kateri gumb je bil pritisnjen. Za najbolj neučakane je tu zgled:

```
Private Sub Form_MouseDown(Button As Integer, Shift As Integer, _
X As Single, Y As Single)
  If Button = 2 Then Form1.PopupMenu mnuPreizkus
End Sub
```

Metoda PopUpMenu lahko prejme še nekaj dodatnih parametrov. Celotna oblika metode je razložena v sprotni pomoči.

Drugi gradniki

Preostali vgrajeni gradniki in tudi zunanji gradniki ActiveX bodo prišli na vrsto, ko bomo reševali praktične zglede. Lotili se jih bomo, brž ko spoznamo še osnovne programske stavke in konstrukte visual basica.

6. POGLAVJE

Programska koda

Programska koda določa *vedenje* nekega programa. Pišemo jo v oknu s kodo. Kakor vemo, pripada okno s kodo vsakemu obrazcu v našem projektu. V projekt je mogoče dodati tudi samostojna okna s kodo, v katerih so splošna programska navodila, neodvisna od uporabniškega vmesnika.

V tem poglavju bomo sistematično spoznali vse člene, ki sestavljajo programsko kodo v visual basicu. Jezik, ki ga uporablja razvojno okolje, je predmetna oblika Microsoftovega basica, ki ima nekaj skladenjskih posebnosti, vendar še vedno ohranja vse značilnosti jezika basic, kot je tolmačeno izvajanje, uporaba nenapovedanih spremenljivk itn. Naštete prednosti basica so za programerja začetnika resda precejšnje olajšanje, vendar skrivajo tudi nekaj pasti. Visual basic je zato nadgrajen z ukazi, ki ga prelevijo v bolj dosleden računalniški jezik, hkrati pa pozna kar nekaj predmetnih razširitev, ki ga postavljajo ob bok najzmogljivejšim programerskim orodjem. V peti različici so ga oplemenitili še s prevajalnikom v strojno kodo, ki izdatno pospeši izvajanje zaokroženih programov.

Ker smo še vedno v prvem delu, ki obravnava osnove visual basica, se s pravim predmetnim programiranjem zaenkrat še ne bomo ukvarjali, čeprav bomo uporabljali že pripravljene predmete. V resnici je v visual basicu težko napisati program, ki bi ne uporabljal predmetov. Navsezadnje je tudi vsak gradnik *predmet*.

Elementi kode

Programska koda je sestavljena iz stavkov (*statements*). Ponavadi je v visual basicu vsaka vrstica samostojen programski stavek, zato ne pozna posebnega ločila, ki bi označeval njegov konec. Vsak programski stavek je sestavljen iz različnih elementov.

Simbolična imena

Med načrtovanjem vmesnika in pisanjem programske kode je treba poimenovati kup stvari. Spomnimo se samo lastnosti Name, ki jo pozna vsak gradnik. Imena predpisujemo tudi *konstantam*, *spremenljivkam*, *podprogramom*, *predmetom* itn. Poimenovati moramo vsak element, ki bi ga radi uporabili v programski kodi (ali, v računalniškem žargonu, na katerega bi se radi *sklicevali*). Večina programskih

jezikov vsiljuje nekaj omejitev, ki jih mora spoštovati vsako *pravilno* simbolično ime (*regular identifier*). Za visual basic so ta pravila naslednja:

- Ime se mora začeti s *pravilno* črko. Pravilna črka je črka angleške abecede ali podčrtaj (_).
- Ime ne sme vsebovati presledka, pike ali posebnih znakov, ki določajo podatkovni tip (to so znaki `%`, `&`, `!`, `#`, `@`, `$`). Tak znak lahko prepišemo le na koncu imena in s tem implicitno določimo podatkovni tip[16].
- Ime ne sme biti daljše od 255 znakov, imena gradnikov pa ne smejo presegati dolžine 40 znakov.
- Ime ne sme biti enako *rezervirani besedi*.

Rezervirane besede (*keywords*) so sestavni del programskega jezika, to so npr. besede **Private**, **Sub**, **If**. Ker imajo natančno določen pomen, ni dopustno, da bi jih uporabili še za kaj drugega. Urejevalnik programske kode jih samodejno obarva (po privzeti nastavitvi modro).

> **NAMIG** Gradnike je mogoče poimenovati tudi s simboličnim imenom, ki je enako rezervirani besedi, čeprav to ni priporočljivo. V tem primeru moramo ime v programski kodi uporabiti tako, da ga oklepata oglata oklepaja (`[]`). Ker je visual basic še vedno živ jezik, ki mu razvijalci dodajajo nove rezervirane besede, se lahko zgodi, da bo danes pravilno simbolično ime jutri prepovedano (čeprav je to zelo malo verjetno). Z opisanim postopkom bo tedaj hitro mogoče popraviti programsko kodo.

Oglejmo si nekaj zgledov pravilnih in nepravilnih simboličnih imen:

Nepravilno	Pravilno	Razlog
črka	crka	Uporaba znakov iz neangleških abeced ni priporočljiva, čeprav ni izrecno prepovedana.
1oseba	oseba1	Prvi znak ne sme biti številka.
oddano&predano	oddanoInPredano	V imenu je prepovedan znak.
for	_for	Pravilno ime ne sme biti enako rezervirani besedi.

Ker visual basic *ne* ločuje med malimi in velikimi črkami, je vseeno, kako zapišemo simbolično ime. Če želimo izboljšati berljivost kode, uporabljamo kombinacije velikih in malih črk ter besede ločujemo z znakom podčrtaj (_). Urejevalnik

[16] Uporaba posebnih znakov za implicitno določanje podatkovnega tipa je ostanek preteklosti. Take uporabe ne priporočajo več.

programske kode samodejno spreminja velikost znakov za rezervirane besede, hkrati pa skrbi tudi za enako izpisovanje naknadno določenih simboličnih imen.

Konstantne vrednosti

Nespremenljive podatke zapišemo kot konstantne vrednosti (*literals*). V visual basicu lahko tako zapišemo cela in realna števila, datume, logične vrednosti in nize znakov.

Cela števila

Cela števila najpogosteje zapisujemo v desetiški obliki. Tu in tam je bolj priročno, če uporabimo osmiški ali šestnajstiški zapis, ki ju visual basic tudi dopušča. Celo število v osmiški obliki zapišemo tako, da predenj postavimo zaporedje `&O` (o kot *octal*), v šestnajstiški pa zaporedje `&H` (h kot *hexadecimal*). Nekaj zgledov:

Desetiško	Osmiško	Šestnajstiško
1	&O1	&H1
10	&O12	&HA
42	&O52	&H2A

Šestnajstiški zapis je posebej uporaben za določanje barve. Za zgled si lahko ogledamo, kako je zapisana nastavitev lastnosti BackColor.

> **NAMIG** Cela števila lahko zapišemo z znakom & na koncu, ki prevajalniku naloži, naj ga obravnava kot dolgo, 32-bitno celo število (tip Long, ki ga bomo še spoznali).

Realna števila

Realna števila vedno zapisujemo v desetiški obliki z decimalnim ločilom. Decimalno ločilo je vedno pika (.), ne glede na krajevne nastavitve. Na voljo imamo tudi dve obliki eksponentnega zapisa. Nekaj zgledov:

Navadno	Eksponentno v enojni natančnosti	Eksponentno v dvojni natančnosti
0.001	1E-3	1D-3
3.14	3.14E0	3.14D0
1000023	1.000023E6	0.1000023D7

Eksponentni zapis določa mantiso in desetiški eksponent, ki sta oba lahko negativni ali pozitivni števili. Urejevalnik kode bo nekatere eksponentno zapisane konstante samodejno pretvoril v desetiško obliko (take so vse zgoraj naštete). Uporabno razliko med zapisoma v različni natančnosti bomo spoznali, ko bomo obravnavali podatkovne tipe.

Datum

Datumske vrednosti lahko zapišemo kot *datumski niz*. Datumski niz (*date literal*) je zaporedje znakov, ki jih oklepata znaka # (*hash*). Datumsko vrednost v kodi moramo zapisati ustrezno angleškim krajevnim nastavitvam (enako velja tudi za decimalna števila), uporabniki pa lahko datumske vrednosti vnašajo v vnosna polja tudi v krajevni obliki, ki je določena s sistemskimi nastavitvami. Nekaj zgledov iz kode:

```
dan = #1/13/1998#
dan = #jan/13/98#
dan = #13/jan#
```

Urejevalnik kode bo vse zapisane zglede samodejno pretvoril v obliko prvega zapisa (če smo še vedno v letu 1998). V datumsko vrednost lahko zapišemo tudi uro dneva:

```
trenutek = #jan/13/98 15:00#
trenutek = #13/jan 3PM#
```

Kar bo urejevalnik kode samodejno pretvoril v:

```
trenutek = #1/13/98 3:00:00 PM#
```

> **NAMIG** Kakor bomo še videli, lahko datumske vrednosti obravnavamo tudi kot realna števila. Trenutek, ki smo ga obravnavali, predstavlja tudi realno število 35.808,63.

Logične vrednosti

Ko zapisujemo konstantne logične vrednosti, imamo na voljo le resnico in neresnico. Za vsako je rezervirana ustrezna beseda:

Vrednost	Rezervirana beseda
Neresnica	**False**
Resnica	**True**

> **NAMIG** V logičnih izrazih lahko False nadomestimo z 0, True pa s katerimkoli celim številom razen 0. Če želimo, da bi pravilno deloval tudi logični operator Not, True predstavimo s celim številom –1.

Dejansko omenjeni rezervirani besedi v prvi inačici visual basica niti nista bili določeni.

Znaki

Posamezen znak ali nize znakov zapišemo v visual basicu z dvojnimi narekovaji ("). Če želimo v niz vključiti tudi znak za dvojni narekovaj, ga zapišemo dvakrat. Spodnji zgled:

```
MsgBox "Pravijo mi ""bistri""!"
```

prikaže okno z naslednjim besedilom:

Slika 6-1: Prikaz niza znakov

Ločila

Omenili smo že, da v visual basicu ne poznamo posebnega ločila, ki bi označeval konec programskega stavka, ker je vsak stavek zapisan v svoji vrstici. To je tudi priporočljiv način kodiranja. Vseeno znajo biti posamezni programski stavki precej dolgi, zato je na voljo posebno ločilo, ki omogoča zapis stavka v več vrsticah. Ločilo sestavljata presledek in podčrtaj (_), ki ju zapišemo na koncu vrstice, ki se nadaljuje:

```
MsgBox "Lep pozdrav " & _
  Text1.Text & _
  ", kako je kaj danes?"
```

Oznake za nadaljevanje programske vrstice seveda ne moremo zapisati znotraj konstantnega niza znakov, zato smo več nizov združili z *operatorjem* &. Velja tudi, da oznaki za prehod v novo vrsto ne sme slediti *komentar*.

Visual basic omogoča tudi zapis več stavkov v isti vrstici. Ločimo jih z dvopičjem (:):

```
sporocilo = Text1.Text: MsgBox sporocilo
```

Med ločila uvrščamo tudi označevalec datumskih nizov (#), dvojni narekovaj, par zavitih ali oglatih oklepajev – njihovo vlogo bomo še natančneje spoznali – ter enojni narekovaj, ki začenja *komentar*.

Komentarji

Komentarji pojasnjujejo programske stavke in se med izvajanjem programa nikakor ne upoštevajo. Komentarje je priporočljivo vnašati pri vsakem bolj zapletenem

projektu, še posebej pri programih, ki jih bo treba dlje časa vzdrževati in dopolnjevati.

V visual basicu lahko komentar začnemo na dva načina:

- z rezervirano besedo **Rem**,
- z enojnim narekovajem (').

Prvi način je praktično izumrl, saj predstavlja samostojen programski stavek, medtem ko lahko z uporabo enojnega narekovaja začnemo komentar kjerkoli (edina izjema je nesklenjena vrstica)[17]. V obeh primerih velja, da komentar traja do konca vrstice. Če želimo zapisati večvrstični komentar, moramo vsako vrsto začeti z ustrezno rezervirano besedo ali simbolom:

```
' Odzivni podprogram za
' dogodek CLICK
Private Sub Command1_Click()
  MsgBox Text1.Text ' izpiše vsebino vnosnega polja
End Sub
```

Urejevalnik po privzeti nastavitvi vse komentarje obarva zeleno.

Izrazi in operatorji

Z *izrazi* izračunavamo nove vrednosti ali dobimo dostop do podatkov. Po vrsti izida, v katerega se izraz ovrednoti, jih delimo na računske, logične in sklicne. V vsaki izmed teh vrst izrazov nastopajo *operatorji*, simboli za prirejevalno, računsko, primerjalno, logično ali sklicno operacijo.

Prireditveni operator

Najpogostejši izraz pri programiranju je prireditveni izraz, ki *konstanti* ali *spremenljivki* priredi novo vrednost. Operator prirejanja je v basicu znak enako (=).

```
pozdrav = "Pozdravljen, " & Text1.Text
```

Računski operatorji

Visual basic pozna naslednje računske operatorje:

Operator	Pomen
+	Seštevanje.

[17] Urejevalnik v visual basicu obravnava rezervirano besedo **Rem** kot sinonim za simbol ', zato sta oznaki popolnoma enakovredni.

Operator	Pomen
-	Negacija (–10) ali odštevanje (10 – 20).
*	Množenje.
/	Deljenje.
\	Celoštevilčno deljenje.
Mod	Ostanek pri deljenju.
^	Potenciranje.

Visual basic pri celoštevilčnem deljenju ali pri izračunavanju ostanka vedno računa s celimi števili. Če so v izrazu realna števila, jih prej zaokroži. Zanimivost je operator za potenciranje, ki ga med programskimi jeziki le redko najdemo:

```
koren = 144 ^ 0.5 ' koren = 12
```

> **NAMIG** Za preračunavanje lahko uporabimo tudi bitne operatorje, ki so enaki logičnim. Zgled uporabe bomo spoznali, ko jih bomo obravnavali.

Primerjalni operatorji

Primerjalni izrazi se vedno ovrednotijo v logično vrednost. Visual basic pozna naslednje operatorje za primerjanje:

Operator	Pomen
=	Je enako?
<	Je manjše?
<=	Je manjše ali enako?
>	Je večje?
>=	Je večje ali enako?
<>	Ni enako?
Like	Je podobno?

Visual basic za prirejanje in primerjanje enakosti uporablja enak operator. Za reševanje morebitnih dvoumnosti lahko posežemo po oklepajih, sicer pa vgrajena prioriteta operatorjev natančno določa vsak izraz. Posebnost je operator **Like**.

Omogoča primerjanje besedil na podlagi *vzorca*. Vzorec (*pattern*) je niz znakov, v katerem lahko nastopajo nekateri posebni simboli in seznami:

Kombinacija	Pomen
`?`	Katerikoli znak.
`*`	Katerokoli, tudi prazno zaporedje znakov.
`#`	Katerakoli števka.
`[seznam]`	Katerikoli znak s seznama.
`[!seznam]`	Katerikoli znak, ki ni na seznamu.

Seznam znakov je lahko sestavljen iz več znakov ali iz intervala, ki ga določimo z vezajem (–). Nekaj zgledov uporabe operatorja **Like**:

```
MsgBox "Vedno" Like "V*"         ' True
MsgBox "Vedno" Like "V?"         ' False
MsgBox "Vedno" Like "V????"      ' True
MsgBox "Vedro" Like "Ved[rn]o"   ' True
MsgBox "Vedno" Like "Ved[a-z]o"  ' True
MsgBox "Vedno" Like "*o"         ' True
```

NAMIG Na primerjanje nizov pomembno vpliva ukaz **Option Compare**. Stavek zapišemo za vsak modul posebej v del splošnih napovedi (*general declarations*). Splošna oblika stavka je:

```
Option Compare {Binary|Text}
```

Če stavka ne zapišemo, se privzame *binarno* primerjanje. Binarno primerjanje (*binary*) primerja znak za znakom na podlagi njegove kode. Znaka A in a nista enaka, krajevne nastavitve pa nimajo nobenega vpliva. Primerjanje besedila (*text*) primerja znake ne glede na velikost črk. V tem primeru sta znaka A in a enaka, pravilno pa se upoštevajo tudi krajevne nastavitve, tako da je npr. znak č »manjši« od d in »večji« od A.

Logični operatorji

V izrazih lahko nastopata tudi logični vrednosti, ki ju lahko preračunavamo po Boolovi algebri z naslednjimi operatorji:

Operator	Pomen
`Not`	Logični ne.
`And`	Logični in.

Operator	Pomen
`Or`	Logični ali.
`Xor`	Logična neenakost.
`Eqv`	Logična enakost.
`Imp`	Logična implikacija.

NAMIG Za vse logične izračune po pravilih Boolove algebre zadoščata že operatorja **Not** in eden izmed **And** in **Or**. Te tudi najpogosteje uporabljamo.

Z logičnimi operatorji najpogosteje povezujemo primerjalne izraze, kakor v spodnjem zgledu:

```
pogoj = ((delo <= 35) And (placa > 200000)) _
  Or (placa \ delo > 6000)
```

Logične operatorje lahko uporabimo tudi za bitno preračunavanje s celimi števili. V tem primeru bo operator izvedel logično operacijo na vsakem paru bitov navedenih števil (ali na posameznem bitu pri operatorju **Not**), izid pa je novo število, zato je tak izraz pravzaprav računski:

```
MsgBox Not 0         ' -1
MsgBox &HF0 And &H10 ' 16
```

Združevanje nizov

Za združevanje nizov imamo na voljo dva operatorja: + in &. Prvi je na voljo zgolj zaradi združljivosti s starejšo kodo in njegove rabe ne priporočamo. Še posebej kočljivo ga je rabiti v izrazih, v katerih nastopajo prilagodljive spremenljivke. Oglejmo si dva zgleda:

```
s = "13"
n = 2
s = s + n        ' s = 15
s = "K" + "" & s ' s = "K15"
```

```
s = "13"
n = 2
s = s & n        ' s = "132"
s = "K" & s      ' s = "K132"
```

Več o prilagodljivosti spremenljivk bomo izvedeli, ko bomo obravnavali podatkovne tipe.

Prioriteta operatorjev

Na vrstni red izračunavanja izrazov lahko vplivamo z oklepaji. Če teh ni, vrstni red določa vgrajena prioriteta operatorjev. Absolutno prednost imajo računski operatorji, sledijo jim primerjalni, na koncu pa pridejo na vrsto še logični. V celoti je vrstni red – od prvega do zadnjega – naslednji:

1. Potenciranje (^).
2. Nasprotna vrednost (-).
3. Množenje, deljenje (*, /).
4. Celoštevilčno deljenje (\).
5. Ostanek pri deljenju (Mod).
6. Seštevanje, odštevanje (+, -).
7. Združevanje nizov (&) .
8. Enakost (=).
9. Neenakost (<>).
10. Je manjše (<).
11. Je večje (>).
12. Je manjše ali enako (<=).
13. Je večje ali enako (>=).
14. Primerjanje vzorcev (Like).
15. Logični ne (Not).
16. Logični in (And).
17. Logični ali (Or).
18. Logična neenakost (Xor).
19. Logična enakost (Eqv).
20. Logična implikacija (Imp).

Če imata operatorja enako prioriteto, se izraz izračuna od leve proti desni.

Podatki in vrste

Podatke, s katerimi program dela, lahko zapisujemo kot konstantne vrednosti, vendar bi bil tak program komajda uporaben. Programi podatke v glavnem obdelujejo, zato jih predstavljamo v obliki *spremenljivk*. Spremenljivka (*variable*) je simbolična oznaka spremenljive vrednosti, ki je lahko število, logična vrednost, besedilo in tudi predmet. Torej spremenljivko v celoti določajo njeno ime, njena trenutna vsebina, po želji pa ji lahko predpišemo tudi *vrsto*. Vrsta spremenljivke (*data type*) omejuje vrste vrednosti, ki jih spremenljivka lahko hrani. Spremenljivka celoštevilčnega tipa tako ne more shranjevati npr. besedila.

V programih se določene konstantne vrednosti pogosto ponavljajo. Za njihovo shranjevanje bi sicer lahko uporabili spremenljivko, vendar imamo na voljo še priročnejšo shrambo: *konstanto*. Konstanta (*constant*) je simbolična oznaka nespremenljive vrednosti. Tudi konstanta ima lahko predpisan podatkovni tip.

Napoved

Programski jezik basic ne zahteva izrecnega napovedovanja spremenljivk, ki jih bomo uporabili v programski kodi. To lastnost je podedoval tudi visual basic. V odzivni proceduri dogodka Click bi tako brez težav zapisali:

```
Private Sub Command1_Click()
  besedilo = Text1.Text
  MsgBox besedlo
End Sub
```

Slika 6-2: V čem je težava?

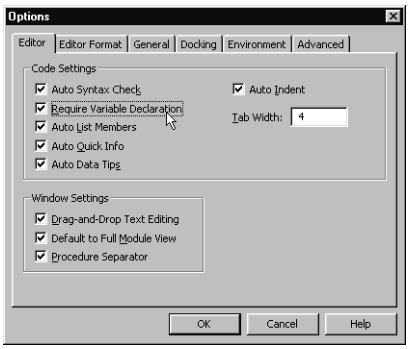
Za izpis vsebine vnosnega polja smo uporabili spremenljivko, imenovano *besedilo*. Najprej smo ji priredili vsebino vnosnega polja, zatem pa smo jo uporabili kot parameter podprogramu MsgBox. Če program poženemo, je okno, ki ga prikaže podprogram MsgBox, vedno *prazno*, ne glede na to, kaj smo vnesli v vnosno polje. V čem je težava?

Vse spremenljivke imajo *privzeto* vrednost, ki je število 0, prazno besedilo, neresnica ali neveljaven predmet, odvisno pač od tega, kako spremenljivko obravnavamo. V zgornjem zgledu smo zagrešili čisto navadno tipkarsko napako. Ime spremenljivke smo drugič zapisali kot `besedlo` namesto `besedilo`. Za visual basic je bila to nova spremenljivka, priredil ji je prazen niz, ki ga MsgBox tudi (ne) prikaže.

Tovrstne težave nas lahko stanejo precej živcev, a so povsem nepotrebne. Visual basic nam ponuja, da bo sam preverjal napoved vseh spremenljivk. Če to možnost izkoristimo, resda zapravimo delček udobja, ki ga ponuja raba nenapovedanih spremenljivk, vendar so prednosti neprimerljivo večje. Če se zavedamo, da nenapovedanih spremenljivk zunaj podprogramov tako ali tako ne moremo uporabljati, pa je že kar čudno, da samodejno preverjanje napovedi spremenljivk ni vključeno že od vsega začetka[18].

Prvo, kar napravimo pred obširnejšim kodiranjem, je zato vklop preverjanja napovedi. Dosežemo ga z izbiro **Tools** ➪ **Options** ➪ jeziček **Editor** ➪ vključimo stikalo **Require Variable Declaration**.

[18] Vzrok je v združljivosti s staro kodo.

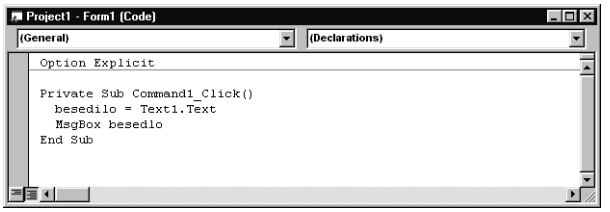


Slika 6-3: Vključeno preverjanje napovedovanja spremenljivk

Zelo pomembno: z vklopljenim preverjanjem napovedi bo visual basic v vsako *novo* okno s kodo dodal programski stavek

```
Option Explicit
```

v splošni del modula (*general declarations*). Obstoječa okna s kodo se ne bodo prav nič spremenila, zato moramo vanje omenjeni stavek dodati ročno. To postorimo tako, da v oknu s kodo z levega seznama predmetov izberemo splošni del (*General*), z desnega seznama pa napovedi (*Declarations*), zatem vnesemo programski stavek:



Slika 6-4: Ročno vnašanje stavka, ki vklopi preverjanje napovedi.

Za vse nove, iz nič začete projekte, nam ni treba skrbeti. Stavek **Option Explicit** se samodejno doda na vrh vsakega modula.

Tako popravljen zgled lahko poženemo na običajni način. Prevajalnik se sicer ne bo pritožil, vendar nas bo, ko bomo kliknili gumb, pričakalo sporočilo o napaki. Bolje je, če program poženemo z vnaprejšnjim popolnim prevajanjem. To dosežemo tako, da v menuju **Run** izberemo ukaz **Start With Full Compile** ali pa pritisnemo bližnjico Ctrl+F5. Prevajalnik se bo sedaj pritožil takoj in programa ne bo niti poskušal izvajati.

Zdaj moramo vsako spremenljivko pred uporabo izrecno *napovedati* (*declare*). To lahko storimo na naslednji način:

```
Dim imeSpremenljivke [As podatkovniTip]
```

Kakor vidimo, podatkovnega tipa ni treba zapisati. V istem stavku Dim lahko napovemo tudi več spremenljivk, vendar moramo za vsako predpisati podatkovni tip:

```
Dim a, b
Dim i, j, k As Integer
```

V zgornjem zgledu je samo `k` vrste Integer, vse druge spremenljivke so vrste Variant. Sama napoved spremenljivke že zasede pomnilniški prostor, potreben za shranjevanje njene vsebine. Napoved tudi postavi to vsebino na privzeto vrednost, odvisno od izbranega podatkovnega tipa.

> **NAMIG** Čeprav je lepa programerska navada, da vse uporabljene spremenljivke napovemo na vrhu podprograma oziroma modula, nam tega visual basic ne vsiljuje. Ko je vklopljeno preverjanje, zadošča, da spremenljivko napovemo, tik preden jo uporabimo. Stavek Dim je moč zapisati kjerkoli *znotraj* podprograma.

Doseg in življenjska doba

Spremenljivke in konstante v posameznem oknu s kodo (*modulu*) lahko napovemo na dveh mestih:

- Znotraj podprograma. Taka spremenljivka ali konstanta je dosegljiva in živi samo znotraj podprograma.
- V delu splošnih napovedi (*general declarations*). Taka spremenljivka ali konstanta je dosegljiva vsem podprogramom modula in živi, dokler živi modul.

Mesto napovedi torej vpliva na *doseg* in *življenjsko dobo*. Doseg (*scope*) določa mesta v kodi, na katerih lahko napovedano spremenljivko ali konstanto uporabimo. Pravilno ime spremenljivke oziroma konstante mora zato zadoščati še enemu pogoju: biti mora *enolično* v vsem njenem dosegu. Različen doseg nam omogoča, da znotraj več procedur uporabimo enako poimenovane spremenljivke oziroma konstante, ki so med seboj nevidne in neodvisne. V visual basicu imamo tri različne dosege:

- Podprogramski doseg (*local scope*).
- Modulski doseg (*module scope*).
- Globalni doseg (*global scope*).

Življenjska doba (*lifetime*) določa čas, v katerem napovedana spremenljivka hrani vanjo zapisano vrednost. Na doseg in življenjsko dobo vpliva tudi oblika napovedi. Čisto splošno se napoved spremenljivke opisno zapiše takole:

```
{Public|Private|Dim|Static} imeSpremenljivke [As podatkovniTip]
```

Napoved konstante pa ima naslednjo obliko:

```
[[Public|Private]] Const imeKonstante [As podatkovniTip] = izraz
```

Kakor vidimo, moramo konstanti ob napovedi prirediti začetno vrednost, ki je izid *izraza*. Izraz je lahko tudi konstantna vrednost, ni pa nujno:

```
Const minSekund = 60
Const uraSekund = 60 * minSekund
```

Različne kombinacije dosega in življenjske dobe bomo spoznali spodaj.

Napoved v podprogramu

Spremenljivke in konstante, napovedane v podprogramu, so vedno dosegljive le znotraj podprograma. Spremenljivke lahko napovemo samo z rezerviranima besedama **Dim** ali **Static**, konstante pa samo z besedo **Const**. Vse druge kombinacije so znotraj podprograma prepovedane.

Na doseg torej ne moremo vplivati, očitno pa lahko nekaj storimo z življenjsko dobo spremenljivk. Spremenljivke, napovedane z **Dim**, so dinamične in ob *vsakem* izvajanju podprograma najprej dobijo privzeto vrednost. Spremenljivke, napovedane s **Static**, so statične in samo ob *prvem* izvajanju podprograma dobijo privzeto vrednost. Njihova življenjska doba je zato enakovredna modulskim spremenljivkam, ohranjajo pa omejeno dosegljivost.

Zgled:

```
Private Sub Command1_Click()
Const prirastek = 10
Dim umira
Static zivi
  umira = umira + prirastek
  zivi = zivi + prirastek
  MsgBox "umira: " & umira & _
    ", zivi: " & zivi
End Sub
```

Po osmih klikih gumba je izid podprograma MsgBox naslednji:

Slika 6-5: Statična spremenljivka ohranja svojo vrednost

Statična in dinamična spremenljivka sta začeli življenje na nični vrednosti. Z vsakim novim klicem podprograma je dinamična spremenljivka znova začela z ničle, statična pa je ohranjala svojo vrednost, ki se je nepretrgoma povečevala.

Napoved v modulu

Vsako okno s kodo je *programski modul*. Programski modul (*module*) pripada vsakemu obrazcu, v projektu pa imamo lahko tudi neodvisne *kodne* module (*code module*), ki nimajo pripadajočega uporabniškega vmesnika. Tovrstne module uporabljamo za shranjevanje bolj splošnih podprogramov, še posebej primerni pa so za napovedovanje globalnih konstant in spremenljivk.

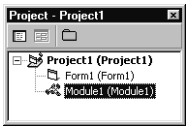
Nov neodvisni kodni modul lahko v projekt dodamo na več načinov:

- Izberemo ukaz **Add Module** iz glavnega menuja **Project**.
- V priročnem menuju projektnega okna izberemo **Add ➪ Module**.
- Za dodajanje novih elementov projekta je na voljo tudi drugi gumb v vrstici gumbov pod glavnim menujem.

Če se nam, ko izberemo katerikoli ukaz, odpre galerija predlog, izberemo ikono **Module** in gumb **Open**. Galerijo je mogoče s stikalom pri dnu tudi izklopiti.

Modul s kodo lahko prikličemo s projektnim oknom, v katerem je prikazan s posebno sličico:

Slika 6-6: Modul s kodo je ponazorjen v projektnem oknu.



Nezaželeni element projekta je mogoče odstraniti na več načinov. Najenostavneje je iz njegovega priročnega menuja izbrati ukaz **Remove**, isti ukaz je na voljo tudi v glavnem menuju **Project**. Pred odstranjevanjem moramo nezaželeni modul najprej označiti v projektnem oknu.

> **NAMIG** Pri dodajanju modula lahko izberemo tudi že prej izdelan in shranjen modul. V priročnem menuju izberemo ukaz **Add File...** ali pa v galeriji izberemo jeziček **Existing**. Kodni moduli so shranjeni v datotekah s podaljškom .BAS.

Na doseg spremenljivk in konstant vpliva vrsta modula, v katerem je njihova napoved. Za začetek si oglejmo modulsko dosegljivost. Vsak modul pozna del s splošnimi napovedmi, v katerega zapišemo napovedi *modulskega* in *globalnega* dosega. Modulski doseg določa spremenljivke in konstante, ki so vidne povsod znotraj istega modula, globalni doseg pa nima omejitev. Globalni doseg za konstante in spremenljivke nekaterih posebnih vrst (omejeni nizi in polja) je mogoče zapisati le v neodvisnem kodnem modulu.

Spremenljivko lahko napovemo z globalnim:

```
Public imeSpremenljivke [As tipSpremenljivke]
```

ali modulskim dosegom:

```
{Private|Dim} imeSpremenljivke [As tipSpremenljivke]
```

Za modulski doseg sta obe rezervirani besedi enakovredni, priporočajo pa uporabo rezervirane besede **Private**.

Spremenljivke dveh posebnih vrst in konstante z globalnim dosegom je mogoče določiti le v neodvisnem kodnem modulu. Globalno konstanto napovemo kot:

```
Public Const imeKonstante [As podatkovniTip] = izraz
```

modulsko konstanto pa:

```
[Private] Const imeKonstante [As podatkovniTip] = izraz
```

Rezervirana beseda **Private** ni obvezna, vendar precej izboljša berljivost kode.

Zelo pomembno: medtem ko so globalne spremenljivke in konstante, napovedane v modulu s kodo, dostopne brez omejitev, so globalne spremenljivke obrazca navzven dosegljive le posredno, prek sklicevanja na obrazec. Oglejmo si zgled:

```
' Koda, ki pripada neodvisnemu modulu
Option Explicit
Public Const besedilo = "Modul!"

Sub IzpisBesedila()
  ' globalna konstana je dosegljiva neposredno
  MsgBox besedilo  ' Modul!
  ' posredno sklicevanje:
  MsgBox Form1.besedilo ' Obrazec!
End Sub

' Koda, ki pripada obrazcu z imenom Form1
Option Explicit
Public besedilo

Private Sub Command1_Click()
  besedilo = "Obrazec!"
  ' spremenljivka ima enako ime in zakrije konstanto
  MsgBox besedilo ' Obrazec!
  IzpisBesedila
End Sub
```

Kakor vidimo, sta globalna spremenljivka obrazca in globalna konstanta poimenovani enako. To je dopustno, saj imata v bistvu različen doseg. Konstanto lahko uporabljamo neposredno, samo z njenim imenom, na spremenljivko pa se moramo zunaj obrazca sklicevati posredno. Drugače je znotraj obrazca. Znotraj podprograma Command1_Click je spremenljivka besedilo vidna, zato zakrije globalno konstanto. Na splošno zaradi boljše berljivosti kode priporočajo, da za globalne konstante uporabljamo enolična imena, ki se sicer v kodi ne ponovijo.

Dejansko smo v zadnjem zgledu obrazcu dodali novo lastnost (*property*)!

> **NAMIG** Kaj pa sam obrazec in gradniki na njem? Vsi so globalni, kakor da bi bili napovedani z rezervirano besedo **Public**. Na obrazec in posredno na njegove gradnike se lahko sklicujemo kjerkoli iz programske kode.

Visual basic pozna cel kup vnaprej določenih globalnih konstant. Njihova imena se začnejo s kratico vb, uporabimo jih lahko npr. pri podprogramu MsgBox:

```
MsgBox "Zaznal sem virus. Izbrišem disk?", _
    vbCritical + vbAbortRetryIgnore + vbDefaultButton3, _
    "Panika!"
```

Vgrajene konstante precej izboljšajo berljivost programske kode. Imajo še eno prednost: če jih dosledno uporabljamo, se lahko njihova vrednost pozneje tudi spremeni, program pa bo še vedno deloval.

Tipi

Spremenljivka ali konstanta lahko hrani podatke različnih vrst: števila, logični vrednosti (resnica, neresnica), besedilo in še kaj. Če ob napovedi določimo podatkovni *tip*, s tem omejimo vrsto vrednosti, ki so lahko shranjene. S tem pridobimo predvsem dvoje:

- Namen spremenljivke je bolj jasno izražen, s tem pa je posledično zmanjšana možnost njene napačne rabe.
- Program je bolj učinkovit.

Podatkovni tip ni samo domena spremenljivk in konstant. Tudi gradniki, lastnosti, funkcije, parametri imajo podatkovni tip. Če podatkovnega tipa ne predpišemo, se privzame prilagodljivi tip Variant. Kodiranje je zato udobnejše, vendar izgubimo prej naštete prednosti. O prilagodljivem podatkovnem tipu več v nadaljevanju, sedaj pa si oglejmo osnovne podatkovne tipe.

Številski tipi

Visual basic pozna pester nabor številskih tipov. Za cela števila so na voljo tipi Byte, Integer in Long. Če vemo, da bo neka spremenljivka hranila zgolj celoštevilčne vrednosti, se še kako izplača uporabiti predpisani tip. Visual basic naravnost blesti pri preračunavanju s celimi števili. Prevedeni programi, ki preračunavajo cela števila, se po hitrosti prav nič ne ločijo od tistih, razvitih npr. v C++. Značilnosti celoštevilskih tipov so naštete v naslednji tabeli.

Tip	Opis	Obseg
Byte	8-bitno celo število brez predznaka	0–255.
Integer	16-bitno celo število, predstavlja ga tudi znak %	–32.768–32.767.
Long	32-bitno celo število, predstavlja ga tudi znak &	–2.147.483.648–2.147.483.647.

Celoštevilski tip Long je zato mogoče določiti na dva načina:

```
Dim l As Long
Dim l&
```

Drugega načina ne priporočajo več in je na voljo zaradi združljivosti s staro kodo. Za tip Byte, ki je bil vpeljan naknadno, te možnosti sploh ni.

NAMIG Med celoštevilske tipe lahko pogojno uvrstimo tudi logični tip Boolean, ki ga predpišemo spremenljivkam, katerih vrednost bo le logična resnica ali neresnica. V visual basicu velja, da logični resnici (*true*) odgovarja celo število –1, neresnici (*false*) pa 0. Interno se tip Boolean hrani v 16 bitih, kakor tip Integer. Kljub temu mešanja številskih tipov in tipa Boolean ne priporočajo več, saj bo z razvojem jezika vloga posebnega logičnega tipa čedalje bolj izražena.

Pri realnih številih imamo na voljo tri tipe: Single, Double in Currency. Slednji je na voljo za posebej občutljiva preračunavanja denarnih vrednosti, pri katerih ne sme priti niti do najmanjših napak pri zaokroževanju, do kakršnih prihaja pri prvih dveh tipih zaradi pretvorbe desetiške vrednosti v dvojiško[19]. Druge značilnosti realnih tipov so naštete v tabeli:

Tip	Opis
Single	32-bitno realno število za manj natančne izračune, predstavlja ga tudi znak !.
Double	64-bitno realno število za bolj natančne izračune, predstavlja ga tudi znak #.
Currency	64-bitno realno število določene natančnosti na štiri decimalna mesta, predstavlja ga tudi znak @.

NAMIG Posebni podatkovni tip, ki se navzven kaže kot datumska vrednost, v resnici pa je 64-bitno realno število, je tip Date. Celi del števila predstavlja dan, decimalni del pa uro. Obseg podatkovnega tipa pokriva čas od 1. januarja leta 100 do 31. decembra leta 9999.

[19] Tip Currency zato ne pretvarja desetiškega zapisa v običajno dvojiško obliko, ampak vsako desetiško števko kodira s posebno kodo.

Vrednost 0,25 predstavlja 6. uro v soboto, 30. decembra 1899, minila je četrtina »ničtega« dne.

Drugi osnovni tipi

Omenili smo že tipa **Boolean** in **Date**. Prvi spremenljivki predpisuje logično vsebino, ki je lahko zgolj resnica ali neresnica. Čeprav je taka spremenljivka interno predstavljena kot število, je ne moremo uporabljati v številskih izračunih.

Drugače je s spremenljivko tipa Date. Ta sploh ne skriva svojega številskega porekla in jo lahko uporabimo v vseh številskih preračunavanjih. Spremenljivki tega tipa lahko vrednost priredimo v obliki datumskega niza (spoznali smo ga pri obravnavanju konstantnih vrednosti) ali kar neposredno v obliki decimalnega števila. Celi del števila določa dan (negativne vrednosti predstavljajo dneve pred 30. 12. 1899), decimalni del pa uro v dnevu (0 je polnoč, 0,5 pa poldan). Datume je zato zelo preprosto preračunavati.

Pomemben podatkovni tip je *niz* znakov (*string*). Visual basic ločuje med dvema oblikama nizov:

- Nizi spremenljive dolžine (*variable-length strings*).
- Nizi določene dolžine (*fixed-length strings*).

Njuni napovedi sta razvidni iz spodnjega zgleda:

```
Dim niz As String       ' niz spremenljive dolžine
Dim znak As String * 1  ' niz s prostorom za en sam znak
Dim geslo As String * 8 ' niz osmih znakov
```

Niz spremenljive dolžine lahko sprejme skoraj neomejeno število znakov (več kakor 2 milijardi, če prej ne zmanjka pomnilnika), med uporabo se ustrezno vsebini širi in krči. Niz določene dolžine svoje velikosti ne spreminja, največja vnaprej predpisana dolžina je lahko 65.535 znakov. Če mu priredimo vsebino, ki je krajša, se na koncu doda ustrezno število presledkov. Če je vsebina predolga, se njen konec odreže. Nize določene dolžine lahko napovemo globalno le v kodnem modulu, v kodi obrazca pa ne.

NAMIG Nizi spremenljive dolžine zasedejo v pomnilniku 10 zlogov, s katerimi je zapisana dinamična dolžina niza, in še dodaten potreben prostor za shrambo besedila. Nizi nespremenljive dolžine ne zahtevajo dodatnega pomnilnika, zasedejo pa toliko zlogov, kolikor je njihova dolžina.

Naštevni tip

Visual basic nam omogoča, da skupino konstantnih vrednosti združimo v *naštevni tip* (*enumeration*). Ta nam olajša kodiranje, saj poleg obeh že omenjenih prednosti posebnih podatkovnih tipov (trdoživost in učinkovitost kode) uspešno izkorišča pomoč pri pisanju programske kode.

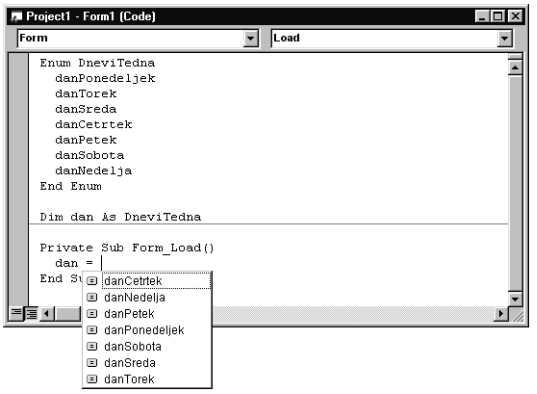
Naštevni tip določimo s stavčnim blokom **Enum**. Naslednji zgled določa sedem konstantnih vrednosti, ki opredelijo vsak dan v tednu:

```
Enum DneviTedna
  danPonedeljek
  danTorek
  danSreda
  danCetrtek
  danPetek
  danSobota
  danNedelja
End Enum
```

V programski kodi lahko sedaj uporabljamo sedem konstant z vrednostmi od 0 do 6, hkrati pa imamo na voljo še nov podatkovni tip. Če napovemo spremenljivko:

```
Dim dan As DneviTedna
```

nam bo pri prirejanju njene vrednosti urejevalnik kode nemudoma priskočil na pomoč, tako kakor je razvidno s spodnje slike:



Slika 6-7: Olajšanje kodiranja pri uporabi naštevnega tipa

Ni nujno, da konstante vrednosti v naštevnem tipu pokrivajo zvezni interval celih števil, ki se začne pri nič. Naš tip bi lahko določili tudi takole:

```
Enum DneviTedna
  danPonedeljek = -1
  danTorek = 4
  danSreda = 8
  danCetrtek    ' pridobi vrednost 9
  danPetek      ' pridobi vrednost 10
```

```
  danSobota     ' pridobi vrednost 11
  danNedelja    ' pridobi vrednost 12
End Enum
```

Vrednosti, ki jih eksplicitno ne določimo, se izračunajo iz nazadnje določene.

Tip Variant

Če spremenljivki ne priredimo posebnega podatkovnega tipa, privzame prilagodljivi tip **Variant**. Da bi bila programska koda lažje berljiva, lahko tip tudi eksplicitno zapišemo. Spremenljivka tega tipa lahko hrani podatke *vseh* osnovnih tipov, primerna pa je tudi za nekatere posebne vrednosti. Najuporabnejša lastnost prilagodljivega tipa je njegovo samodejno prilagajanje izrazom, v katerih nastopa. Pri uporabi prilagodljivega tipa v izrazih moramo paziti na dvoje:

- Če spremenljivko prilagodljivega tipa uporabljamo v računskih izrazih, mora vsebovati nekaj, kar je mogoče tolmačiti kot število.
- Če združujemo nize, moramo uporabljati operator & (spomnimo se obravnave operatorjev za združevanje nizov).

Tip Variant omogoča natančno shranjevanje posebej velikih števil v svojo podobliko Decimal, ki za razliko od drugih številskih tipov ni na voljo kot samostojni podatkovni tip. V tej obliki lahko število zapišemo na 29 mest absolutno natančno, z od 0 do 28 decimalnimi mesti.

> **NAMIG** Tip Variant zaradi svoje prilagodljivosti terja nekoliko več pomnilniškega prostora. Vsaka spremenljivka tega tipa zasede najmanj 16 zlogov, pri nizih znotraj tipa Variant pa 22 zlogov za prazen niz.

Tip Variant lahko vsebuje tudi tri posebne vrednosti: **Empty**, **Null** in **Error**. Za navadne spremenljivke že vemo, da imajo privzeto vrednost, ki je 0, false ali prazen niz, odvisno pač od podatkovnega tipa. Vsaka spremenljivka tipa Variant pa je na začetku prazna, kar označuje posebna vrednost Empty (*prazno*). Ta vrednost izgine, brž ko spremenljivki priredimo neko vrednost (jo *inicializiramo*), tudi če je nova vrednost 0, false ali prazen niz. Če prazno prilagodljivo spremenljivko uporabimo v izrazu, ima vrednost 0, false ali prazen niz, vendar ohranja prazno stanje. Tega lahko preverimo s *funkcijo* IsEmpty. Oglejmo si zgled:

```
Dim n As Variant

If IsEmpty(n) Then n = n + 10
If IsEmpty(n) Then n = 0
MsgBox n     ' prikaže 10
```

Najprej je bil `n` res prazen, vendar smo ga takoj povečali za 10, naslednjič je tako funkcija vrnila neresnično vrednost, zato je `n` ostal na vrednosti 10. Spremenljivko

lahko znova postavimo na prazno vrednost, tako da ji priredimo rezervirano besedo **Empty**:

```
n = Empty
```

Naslednja posebna vrednost v tipu Variant je nedoločena vrednost **Null**. Nedoločena vrednost je največ v rabi pri zbirkah podatkov, v katerih nakazuje neobstoječi zapis. Nedoločena vrednost Null pozna nekaj posebnih zakonitosti:

- Če kjerkoli v izrazu nastopa vrednost Null, je tudi izid izraza nedoločene vrednosti Null.
- Če vgrajeni funkciji posredujemo nedoločen parameter Null in funkcija vrača tip Variant, bo ta funkcija zagotovo vrnila vrednost Null.

Zaradi teh lastnosti pravimo, da se vrednost Null razširja skozi izraze in funkcije. To lastnost lahko uporabimo pri preverjanju napačnega izračuna. Čeprav smo izračunali več izrazov in poklicali več funkcij, zadošča, da preverimo, ali je končni izid enak vrednosti Null. Preverjanje opravi funkcija IsNull, za prirejanje nedoločene vrednosti pa lahko uporabimo ustrezno rezervirano besedo. Prirejanje te vrednosti spremenljivki, ki ni tipa Variant, bo povzročilo napako:

```
' v bo zdaj zagotovo Null
If Not IsNull(v) Then v = Null
```

Tretja posebna vrednost tipa Variant je **Error**, ki ponazarja napako. Z njo lahko znotraj podprograma zaznamo morebitno napako in jo odrazimo navzven, ne da bi prekinili izvajanje programa. Spodnji zgled bo zaznal morebitno napačno deljenje in ga odrazil skozi omenjeno posebno vrednost:

```
Function Deli(a, b)
  If IsNumeric(a) And IsNumeric(b) Then
    If b <> 0 Then
      Deli = a / b
    Else
      ' napaka s kodo 2
      ' deljenje z nič
      Deli = CVErr(2)
    End If
  Else
    ' napaka s kodo 1
    ' napačna parametra
    Deli = CVErr(1)
  End If
End Function

Private Sub Command1_Click()
Dim v
  v = Deli(100, Text1.Text)
  ' je prišlo do napake?
  If IsError(v) Then
    ' da, izpiši njeno kodo
    MsgBox "Napaka s kodo: " & CInt(v)
```

```
  Else
    ' ne, v vsebuje izid deljenja
    MsgBox "Izid deljenja: " & v
  End If
End Sub
```

Funkcija Deli uporabi dve preverjanji. Najprej ugotavlja, ali je posredovana parametra mogoče obravnavati kot število (funkcija IsNumeric). Če se ju da, preveri še delitelja, ki ne sme biti enak 0. Vsaka napaka se odrazi z drugo številsko kodo. Kakor vidimo, je mogoče vrednost Error vpisati v prilagodljivo spremenljivko le s funkcijo CVErr, ki ji kot parameter posredujemo kodo napake. Tovrstno kodo lahko izluščimo le tako, da vrednost s funkcijo CInt pretvorimo v celo število. Na splošno je lahko koda napake tudi realno število, kako ga izluščimo, pa si oglejmo v nadaljevanju.

Preverjanje in pretvarjanje tipov

Pri delu s tipom Variant nam pomaga cel kup funkcij, ki obravnavajo njegove različne podvrste. Dovolj informacij nam ponuja že funkcija VarType, ki vrne celoštevilčno kodo dejanskega tipa, vsebovanega v spremenljivki. Te kode so označene tudi z vgrajenimi globalnimi konstantami. Natančno so opisane v elektronski pomoči omenjene funkcije.

> **NAMIG** Podobno kakor za gradnike tudi za programske stavke velja, da o njih izvemo precej več iz sprotne pomoči. Zadošča, da se postavimo k besedi, ki nas zanima, in pritisnemo tipko F1.

Funkcija VarType nam natančno opredeli podvrsto tipa Variant, na voljo pa so tudi bolj splošne funkcije. Njihovo ime se začne z Is (angl. je), vse pa vračajo logično vrednost. Zbrane so v naslednji tabeli:

Funkcija	Opis
IsEmpty	Preveri, ali spremenljivka vsebuje prazno vrednost.
IsNull	Preveri, ali spremenljivka vsebuje neveljavno vrednost.
IsError	Preveri, ali spremenljivka vsebuje kodo ugotovljene napake.
IsNumeric	Preveri, ali spremenljivka vsebuje številsko vrednost.
IsDate	Preveri, ali spremenljivka vsebuje datumsko vrednost.

Funkcije so zelo uporabne za preverjanje vsebine vnosnih polj. Zgled:

```
If IsDate(Text1.Text) Then _
  MsgBox "Vnesli ste datum " & CDate(Text1.Text)
If IsNumeric(Text1.Text) Then _
  MsgBox "Vnesli ste število " & Text1.TextVnos
```

Že v precej zgledih smo uporabljali funkcije za *pretvarjanje* tipa. Imena tovrstnih funkcij se začnejo s črko C (izvira iz angl. *convert,* pretvori). Naštete so v tabeli:

Funkcija	Vrne tip	Sprejme parameter
CBool	Boolean	Poljuben računski izraz ali niz.
CByte	Byte	Izraz ali niz, ki predstavlja števila v obsegu vrnjenega tipa, morebitni decimalni del se zaokroži.
CCur	Currency	Izraz ali niz, ki predstavlja števila v obsegu vrnjenega tipa.
CDate	Date	Poljuben datumski izraz, ustrezen krajevnim nastavitvam.
CDbl	Double	Števila v obsegu vrnjenega tipa.
CDec	Variant/Decimal	Izraz ali niz, ki predstavlja števila v obsegu vrnjenega tipa.
CInt	Integer	Izraz ali niz, ki predstavlja števila v obsegu vrnjenega tipa, morebitni decimalni del se zaokroži.
CLng	Long	Izraz ali niz, ki predstavlja števila v obsegu vrnjenega tipa, morebitni decimalni del se zaokroži.
CSng	Single	Izraz ali niz, ki predstavlja števila v obsegu vrnjenega tipa.
CVar	Variant	Izraz ali niz, ki predstavlja števila v obsegu tipa Double ali poljuben niz.
CVDate	Variant/Date	Poljuben datumski izraz ali niz, ustrezen krajevnim nastavitvam.
CVErr	Variant/Error	Izraz ali niz, ki predstavlja števila v obsegu 0–65535, morebitni decimalni del se zaokroži.
CStr	String	Poljuben.

Vse funkcije povzročijo napako, če jim znotraj parametra podtaknemo neustrezno vsebino. Posebej zanimivo se vede funkcija CStr. Vsebina, ki jo vrne, je odvisna od posredovane vrednosti:

Posredovana vrednost	Vrnjena vsebina
Tip Boolean	Niz z besedo *True* ali *False*.
Tip Date	Niz z datumom, kot ga določajo krajevne nastavitve.
Nedoločena vrednost Null	Povzroči napako.
Prazna vrednost Empty	Prazen niz ("").
Koda napake Error	Niz, ki vsebuje besedo *Error*, presledek in kodo napake.
Število	Število kot niz znakov.

Funkcija CStr je zato splošno uporabna pri izpisovanju različnih podatkov. Zdaj tudi vemo, zakaj smo lahko funkciji MsgBox v izpis podtikali vse mogoče.

Sestavljeni tipi

Zahtevnejše podatkovne strukture lahko pregledno shranimo v spremenljivko katerega od sestavljenih podatkovnih tipov. Visual basic pozna polja (*arrays*), uporabniško določene sestavljene tipe (*user defined types*), krajše, strukture, in predmete (*objects*).

Polje

Polja omogočajo shranjevanje niza podatkov *istega* tipa v eno samo spremenljivko. Posamezen element polja izločimo na podlagi njegove zaporedne številke (*index*). Z njimi lahko poenostavimo programsko kodo. Doslej smo spoznali že polja *gradnikov* (*control arrays*), ki se od navadnih polj razlikujejo v dveh pomembnih značilnostih:

- Polja podatkov imajo vedno določene *vse* elemente z zaporednimi številkami iz določenega intervala, pri poljih gradnikov pa lahko določeni indeksi manjkajo.
- Vsa polja gradnikov so dinamična in jih spreminjamo s stavkoma Load in Unload. Dinamična polja podatkov spreminjamo drugače.

V visual basicu imajo polja še poseben pomen, saj so edina podatkovna struktura, ki se lahko dinamično povečuje in zmanjšuje med izvajanjem programa. Spreminjanje polj gradnikov bomo spoznali v naslednjem poglavju.

Polja podatkov delimo na statična (*fixed-size arrays*), katerih velikost je določena in se ne spreminja, in dinamična (*dynamic arrays*), katerih velikost se med izvajanjem programa lahko spreminja. Najprej si oglejmo statična polja.

Statična polja

Polje napovemo podobno kakor spremenljivko. Razlika je le v paru oklepajev, znotraj katerega je določen interval zaporednih številk in s tem njegova velikost. Splošna oblika napovedi polja je:

```
{Public|Private|Dim|Static} imePolja([spodnjaMeja To ]zgornaMeja) _
[As podatkovniTip]
```

Velikost polja določimo z intervalom, ki ga lahko predpišemo z zgornjo mejo ali z obema. Vsaka meja je lahko celo število v obsegu tipa Long. Nekaj zgledov napovedi polj:

```
Dim x(10), y(10)                    ' 2 × 11 elementov tipa Variant
Dim osebe(1 To 10) As String * 20   ' 10 omejenih nizov
Static zastavice(-2 To 2) As Boolean ' 5 logičnih vrednosti
```

Zelo pomembno: če spodnje meje ne predpišemo, se privzame vrednost 0. Privzeto *spodnjo* mejo lahko eksplicitno določimo za vsak programski modul posebej. V del splošnih napovedi pred napoved kateregakoli polja zapišemo stavek Option Base, ki ima naslednjo obliko:

```
Option Base {0|1}
```

Na voljo imamo le vrednosti 0 ali 1. Vse napovedi polj, ki sledijo in pri katerih spodnja meja ni predpisana, bodo spoštovale to nastavitev, vendar le v tem modulu. Omenjeni stavek zato ni preveč uporaben, raje se odločimo, da bomo vedno predpisali obe meji intervala.

Že sama napoved statičnega polja *zaseže* ustrezno količino pomnilnika. Vsi elementi se postavijo na privzete vrednosti predpisanega podatkovnega tipa. Ker lahko hitro porabimo velikanske količine pomnilnika, moramo pri uporabi statičnih polj predpisati minimalni interval, ki še zadošča za shranjevanje podatkov pri reševanju določenega problema. Če je ta interval preveč ohlapen, raje izberemo dinamično polje.

Ker je interval zaporednih številk lahko poljubno predpisan, nam pri nadaljnji uporabi polja prideta zelo prav vgrajeni funkciji LBound in UBound, ki vračata spodnjo oziroma zgornjo mejo intervala za posredovano polje. Naslednji zgled se sprehodi čez vse elemente polja, neodvisno od predpisanega intervala:

```
For i = LBound(x) To UBound(x)
  MsgBox x(i)
Next
```

Kakor vidimo, imamo do posameznega elementa dostop tako, da poleg imena spremenljivke zapišemo v oklepaju še njegovo zaporedno številko (*index*).

> **NAMIG** Za sprehod prek vseh elementov polja podatkov, polja gradnikov ali *zbirke* raje uporabljamo stavek For Each, ki ga bomo spoznali v nadaljevanju.

Vsebovana polja

Vemo že, da podatkovni tip Variant lahko obsega vsebino vseh drugih osnovnih tipov. Spremenljivka tipa Variant pa lahko vsebuje celo polje. In seveda, lahko imamo polje spremenljivk tipa Variant, izmed katerih vsaka vsebuje polje drugačnega tipa (morda Variant, ki vsebuje novo polje, katerega elementi so polja ...). Zbegani? Oglejmo si zgled:

```
Dim s(1 To 2) As String
  s(1) = "prvi"
  s(2) = "drugi"
```

```
Dim v(2)
  v(0) = Array(5,2,-12)
  v(1) = s
  v(2) = v

MsgBox v(0)(0)    ' 5
MsgBox v(1)(2)    ' "drugi"
MsgBox v(2)(1)(1) ' "prvi"
```

Verižno sklicevanje, ki ga uporabljamo pri `v(2)`, le ponazarja zmožnosti polja prilagodljivega tipa. Nikakor ne velja tega zgleda posnemati, saj pomeni potencialno nevarno programersko prakso.

Zelo uporabna je funkcija Array, ki prejme poljubno število parametrov in vrne polje, zapisano v tipu Variant. Tovrstno polje ima spodnjo mejo določeno ustrezno stavku Option Base (torej 0 ali kvečjemu 1).

> **NAMIG** V tip Variant ne moremo shraniti polja nizov omejene dolžine ali polja podatkov uporabniško določenega tipa.

Pri delu s polji nam pride prav funkcija IsArray. Vrne resnično vrednost, če je v spremenljivki (ne nujno tipa Variant) vsebovano polje. Resnično vrednost bi vrnili vsi naslednji klici:

```
MsgBox IsArray(s)       ' True
MsgBox IsArray(v)       ' True
MsgBox IsArray(v(0))    ' True
MsgBox IsArray(v(2)(1)) ' True
```

ne pa tudi:

```
MsgBox IsArray(v(1)(1)) ' False
```

Večrazsežna polja

Polja so priročen način shranjevanja večjih količin podatkov v eno samo spremenljivko. Določeni problemi so še učinkoviteje rešeni z uporabo *večrazsežnih* polj, ki jih napovemo s seznamom intervalov. Večrazsežna polja (*multidimensional arrays*) ponavadi uporabimo za zapisovanje matrik (dvorazsežno polje). Polje ima lahko skupaj kar šestdeset razsežnosti.

Za spremljanje stanja na šahovnici bi lahko uporabili naslednje dvorazsežno polje:

```
Dim sahovnica(1 To 8, 1 To 8)
```

Že običajno polje lahko zasede precej pomnilnika, z vsako novo razsežnostjo pa se pomnilniške zahteve strmo povečujejo. Naslednje polje:

```
Dim veliko(6, 1 To 100, -2 To 2)
```

je le trirazsežno, pa že zasede 56.032 zlogov pomnilnika (7 × 100 × 5 elementov tipa Variant po 16 zlogov vsak + 3 × 4 + 20 zlogov za opis polja).

NAMIG Polje takoj zasede 20 zlogov pomnilnika za svoj opis in po 4 zloge za vsako razsežnost, za povrh pa še ustrezno količino pomnilnika za shranjevanje želene količine podatkov. Če je polje vsebovano v tipu Variant, zahteva še 12 zlogov več.

Dinamična polja

Dinamična polja so ena najbolj uporabnih podatkovnih struktur v visual basicu. Ohranjajo vse značilnosti drugih polj, le da jim lahko poljubno spreminjamo velikost. S tem se lahko izognemo potrati pomnilnika, ki jo povzročajo statična polja z maksimizirano velikostjo, saj ta ves čas izvajanja programa zasedajo pomnilnik. Dinamična polja nam omogočajo, da pomnilnik uporabimo le tedaj, ko ga resnično potrebujemo.

Dinamično polje napovemo tako, da izpustimo interval:

```
Dim dinamicnoPolje()
```

S samo napovedjo še ne zasežemo pomnilnika, saj ni znano število elementov v polju. Pred uporabo moramo zato dinamično polje pripraviti za novo vsebino. Tu uporabimo stavek **ReDim**:

```
ReDim dinamicnoPolje(100)
```

Interval polja v stavku ReDim predpišemo popolnoma enako kakor pri napovedi običajnih podatkovnih polj. Kadarkoli želimo, lahko spremenimo interval polja, zgornjo ali spodnjo mejo, celo razsežnost polja, vendar izgubimo vse prej shranjene podatke:

```
Dim dinamicnoPolje() As String
ReDim dinamicnoPolje(1 To 3)
dinamicnoPolje(2) = "Test"
ReDim dinamicnoPolje(1 To 5) ' polje razširimo
MsgBox dinamicnoPolje(2)     ' prazen niz
```

Na srečo imamo na voljo tudi spremembo intervala ob ohranitvi vseh obstoječih podatkov. V tem primeru lahko spremenimo samo *zgornjo* mejo intervala *zadnje* razsežnosti. Za ohranitev obstoječih podatkov stavek ReDim zapišemo z rezervirano besedo **Preserve**:

```
ReDim dinamicnoPolje(2, 5)
dinamicnoPolje(2, 1) = "Test"
ReDim Preserve dinamicnoPolje(2, 2)
MsgBox dinamicnoPolje(2, 1) ' "Test"
```

Dinamično polje smo sicer zmanjšali, vendar je bilo za opazovani element še vedno dovolj prostora. Poskus spreminjanja drugih meja ali razsežnosti ob navzočnosti rezervirane besede Preserve povzroči napako med izvajanjem. Pri spreminjanju velikosti z ohranjanjem podatkov nam znova priskoči na pomoč funkcija UBound.

Pri večrazsežnih poljih lahko funkciji UBound in LBound pokličemo še z drugim parametrom, ki podaja želeno razsežnost. Prva razsežnost je označena z 1. Z naslednjim zgledom bomo razpoložljivo zmogljivost polja razširili za en element[20]:

```
ReDim Preserve dinamicnoPolje(2, UBound(dinamicnoPolje, 2) + 1)
```

Kakor vidimo, je lahko nova velikost polja tudi izid izraza, torej jo lahko izračunamo med izvajanjem programa.

Struktura

Podatke različnih tipov lahko združimo v nov podatkovni tip. Imenujemo ga uporabniško določen podatkovni tip (*user defined type*) ali, krajše, *struktura*. Tudi struktura nam, podobno kakor polje, omogoča, da v eno spremenljivko shranimo več podatkov, vendar so ti pri strukturi lahko različnih osnovnih tipov. To nam pri polju sicer omogoča uporaba prilagodljivega tipa, ki pa ni tako učinkovit (glej obravnavo tipa Variant).

Pred uporabo moramo strukturo najprej določiti. Tu nam pomaga stavek **Type**, katerega oblika je:

```
[Private|Public] Type imeStrukture
  imeElementa [([seznamIntervalov])] As podatkovniTip
  [imeElementa [([seznamIntervalov])] As podatkovniTip]
  …
End Type
```

Strukture lahko napovemo le v delu za splošne napovedi (*general declarations*). Spremljati jih mora tudi pridevnik dostopnosti. Znotraj obrazca je to lahko le Private, splošno uporabne javne strukture moramo določiti v neodvisnih modulih s kodo. Vsak element *mora* spremljati njegov podatkovni tip, pa čeprav je to Variant. Posamezni elementi so lahko tudi polja, dinamična ali statična. Število elementov je poljubno.

Zgled uporabne strukture:

```
Private Type Complex
  r As Double
  i As Double
End Type
```

Čeprav sta oba elementa istega tipa, ju moramo napovedati vsakega posebej.

Po definiciji strukture lahko napovedujemo enostavne spremenljivke ali polja tega tipa. Do posameznega elementa strukture pridemo z znakom pika (.), podobno kakor npr. do elementov (gradnikov) obrazca. Oglejmo si zgled rabe:

```
Private Type Oseba
  ime As String * 20
```

[20] Ker je polje dvorazsežno, je nov »element« dejansko polje treh elementov.

```
  rojen As Date
  podatki() As Byte
End Type

Private zaposleni() As Oseba

Private Sub Command1_Click()
  ReDim zaposleni(9) ' 10 zaposlenih

  ' dostop do elementov preko stavka With
  With zaposleni(4)
    .ime = "Janez Novak"
    .rojen = #7/18/68#
    ReDim podatki(1 To 100) ' 100 zlogov dodatnih podatkov
  End With

  ' posreden dostop s sklicevanjem na strukturo
  MsgBox zaposleni(4).ime & ", rojen: " & zaposleni(4).rojen
End Sub
```

Strukture lahko gnezdimo, že definirano strukturo lahko uporabimo v definiciji druge itn.

Podprogrami

Podprogrami (*subroutines*) so zaokroženi kosi programske kode, ki opravljajo večkrat uporabna opravila. Delimo jih na *procedure* in *funkcije*. Funkcije (*functions*) lahko vračajo vrednosti, zato jih je mogoče uporabiti tudi v izrazih, medtem ko so klici procedur (*procedures*) samostojni programski stavki. Nekateri podprogrami, npr. MsgBox, se lahko vedejo kot procedura ali funkcija. V visual basicu velja, da lahko vsako funkcijo obravnavamo kot proceduro in njeno vrnjeno vrednost zavržemo.

S podprogrami se ukvarjamo že od vsega začetka, odkar smo se spoznali z dogodki. Odzive na dogodke namreč določamo v *odzivnih procedurah*. Tu si bomo ogledali razvoj lastnih podprogramov, ki jih imenujemo tudi *splošni* podprogrami.

Procedure

Splošna oblika napovedi nove procedure je:

```
[[Private|Public]] [Static] Sub imeProcedure ([seznamParametrov])
  teloProcedure
End Sub
```

Kadar v oknu s kodo zapišemo napoved podprograma, urejevalnik samodejno doda njen konec (stavek End Sub).

Ime procedure mora biti pravilno simbolično ime. Tudi procedure imajo svoj doseg, ki je po privzeti nastavitvi globalen, tako da lahko rezervirano besedo Public po želji izpustimo. Zaradi boljše berljivosti kode jo večinoma vseeno zapišemo. Če želimo

proceduro skriti znotraj modula, jo napovemo z rezervirano besedo Private. Take so (na začetku) tudi vse odzivne procedure. Če želimo, da so vse spremenljivke, napovedane znotraj procedure, statične, lahko pridevnik Static dodamo kar sami proceduri, spremenljivke pa napovedujemo, kakor želimo (ali z Dim ali s Static).

Zelo pomembno: Podobno kakor pri globalnih spremenljivkah tudi pri globalnih procedurah velja, da so tiste, definirane znotraj obrazca, navzven dosegljive posredno (prek sklicevanja na obrazec), globalne procedure v modulu s kodo pa so neposredno dosegljive povsod. Z dodajanjem javne procedure obrazcu pravzaprav določimo novo *metodo*!

Za odzivne procedure že vemo, da se kličejo samodejno, ob ustreznem dogodku. Procedure, ki jih določimo sami (*splošne* procedure), moramo poklicati iz programske kode, kar storimo tako, da zapišemo njeno ime in morebitni seznam parametrov. Klic procedure je samostojen programski stavek. Iz kode lahko kličemo tudi dogodkovne procedure. To bomo izkoristili v naslednjem zgledu:

```
Private Sub Command1_Click()
  MsgBox "Vnesli ste " & Text1.Text
End Sub

Private Sub Text1_KeyPress(KeyAscii As Integer)
  If KeyAscii = 13 Then
    KeyAscii = 0
    ' ročno pokličemo kar odzivno proceduro
    Command1_Click
  End If
End Sub
```

Če uporabnik v vnosnem polju pritisne tipko Enter (predstavlja jo koda 13), storimo isto, kakor če bi kliknil gumb. Tako lahko pozneje popravljamo samo eno odzivno proceduro. Učinek tipke Enter bo vedno enak kliku gumba.

> **NAMIG** Na splošno raje zapišemo še lasten podprogram, ki ga kličemo iz obeh dogodkovnih. Tako je namen dogodkovnih podprogramov jasneje izražen, kar olajša poznejše vzdrževanje kode.

Procedura se navadno izteče, ko naleti na stavek End Sub. Izvajanje procedure lahko predčasno prekinemo s stavkom **Exit Sub**.

Funkcije

Funkcija vrača vrednost neposredno[21], zato je njena napoved drugačna od napovedi procedure, na splošno se glasi:

```
[{Private|Public}] [Static] Function imeFunkcije ([seznamParametrov]) _
[As tipFunkcije]
```

[21] Procedura lahko vrednost vrne s parametri, prenesenimi po referenci.

```
  telоFunkcije
End Function
```

Za razliko od procedure funkciji pripada neki podatkovni tip, ki ga vrača. Če ga izpustimo, se privzame prilagodljivi tip Variant. V telesu funkcije je poleg vseh morebitnih parametrov vedno na voljo tudi posebna spremenljivka, ki ima *enako* ime kakor funkcija. Izid funkcije je vrednost te spremenljivke v trenutku, ko se funkcija izteče. Podobno kakor proceduro lahko tudi izvajanje funkcije končamo predčasno, s stavkom **Exit Function**.

Naslednji zgled določa funkcijo, ki varno izračuna kvadratni koren. Stavki, ki določajo vrnjeno vrednost, so zapisani **krepko**:

```
Public Function Koren(v)
  If IsNumeric(v) Then
    If v >= 0 Then
      Koren = Sqr(v)
    Else
      Koren = CVErr(2)
    End If
  Else
    Koren = CVErr(1)
  End If
End Function

Private Sub Command1_Click()
  MsgBox CStr(Koren(Text1.Text))
End Sub
```

Funkcija Koren uporablja in vrne tip Variant, kar nam omogoča varno obravnavanje morebitnih napak, kot sta posredovanje neštevilske vrednosti ali poizkus korenjenja negativnega števila. Klic funkcije ni samostojen programski stavek, ampak sestavni del izraza, ki ga podtaknemo proceduri MsgBox. Funkcijo bi lahko poklicali tudi neposredno. V tem primeru bi se program logično obnašal enako, porabil pa bi za drobec več procesorskega časa.

Zasebna funkcija modula lahko vrača tudi uporabniško definiran tip, kar je pri javnih funkcijah prepovedano:

```
Private Type Test
  a As Integer
  b As Byte
End Type

Private Sub Form_Load()
  MsgBox FnTest.a ' 1
End Sub

Private Function FnTest() As Test
  FnTest.a = 1
  FnTest.b = 2
End Function
```

Funkcija ne more neposredno vračati polja, posredno ga lahko podtaknemo v strukturo.

Posredovanje parametrov

Morebitni parametri podprograma so seznam spremenljivk, ki vplivajo ali odražajo potek izvajanja stavkov v telesu podprograma. Seznam na splošno zapišemo takole:

```
[{ByVal|ByRef}] imeParametra [As podatkovniTip] [, seznamParametrov]
```

Za parameter zadošča, da ga poimenujemo s pravilnim simboličnim imenom. Če želimo, lahko predpišemo tudi podatkovni tip, drugače se privzame prilagodljivi tip Variant. Vsak parameter je znotraj podprograma *enakovreden* vsem njegovim spremenljivkam, zato mora biti njegovo ime enolično določeno. Ker se parameter vede kakor spremenljivka, je pomembno, kaj se zgodi s spremembami njegove vsebine, ko je podprograma konec. Na to vplivata pridevnika **ByVal** oziroma **ByRef**, ki je privzet in ga lahko tudi izpustimo.

Navadno (ali s predpisanim pridevnikom ByRef) se parametri v podprogram prenesejo *po referenci*. To pomeni, da so vse spremembe njegove vsebine *vidne* tudi navzven. Če tega ne želimo, lahko parameter napovemo s pridevnikom ByVal. Parameter se bo prenesel *po vrednosti*, spremembe njegove vsebine pa navzven *ne bodo* vidne. Oglejmo si zgled:

```
Private Sub Pristej10(ByVal poVrednosti, poReferenci)
  poVrednosti = poVrednosti + 10
  poReferenci = poReferenci + 10
End Sub

Private Sub Command1_Click()
Dim a, b
  a = 0
  b = 0
  Pristej10 a, b
  MsgBox "a = " & a & ", b = " & b
End Sub
```

Slika 6-8: Parameter, posredovan po vrednosti, se navzven ne spremeni.

Prenos parametrov po referenci je način, kako lahko podprogram (pravi ali funkcijski) posredno vrne eno ali več vrednosti. Ima pa eno omejitev: pri prenosu po referenci se mora tip dejanskega parametra natančno ujemati z napovedanim v glavi podprograma. Naslednji zgled:

```
Private Sub Form_Load()
Dim s        ' Tip Variant
  s = ""     ' V njem je niz
  Test s     ' Napaka! Neujemanje tipov.
```

```
  MsgBox (s)
End Sub

Private Sub Test(ByRef s As String)
  s = "Besedilo..."
End Sub
```

povzroči napako, saj tip Variant – ne glede na njegovo podvrsto – ne ustreza tipu String. Težavi se lahko na videz izognemo na dva načina:

- Parameter posredujemo po vrednosti.
- Podprogramu namesto spremenljivke posredujemo izraz.

Izraz najlaže določimo tako, da posredovano vrednost objamemo z oklepaji:

```
Test (s)    ' (s) se ovrednoti kot izraz
MsgBox (s)  ' izpiše prazen niz???
```

Vendar v obeh primerih spremembe na parametru navzven *ne* bodo vidne! Edina ustrezna rešitev je, da tip parametra uskladimo z dejansko posredovanim. Če je parameter definiran kot prilagodljivi tip Variant, bo prenos po referenci deloval za vse osnovne tipe. Če zgornji zgled po podatkovnih tipih obrnemo:

```
Private Sub Form_Load()
Dim s As String
  Test s
  MsgBox (s)  ' "Besedilo..."
End Sub

Private Sub Test(s)
  s = "Besedilo..."
End Sub
```

deluje. Spremembe prilagodljivega tipa se odrazijo navzven tudi na tipu String. Zapomnimo si, da bodo spremembe navzven vidne le, če bomo spremenljivke v klicu podprograma zapisali neposredno. Pazimo tudi na odvečne oklepaje.

Rekurzija

Visual basic dopušča *rekurzijo*. Rekurzija (*recursion*) omogoča podprogramom, da kličejo sami sebe. To je zelo učinkovita, vendar tudi nevarna programerska tehnika. Preden poženemo podprogram, ki kliče samega sebe, natančno preverimo, ali se bo klicanje kdaj izteklo. V nasprotnem čaka naš program bolj klavrn konec. Naslednji zgled uporablja rekurzijo za izračunavanje fakultete[22]:

```
' Funkcija prejme samo parameter, ki je dovolj
' majhno celo število
Private Function Fakulteta(n As Byte) As Long
  ' pogoj za konec rekurzije
```

[22] Fakulteta števila *n* je matematična operacija, ki izračuna produkt vseh naravnih števil do vključno števila *n*.

```
  If n = 1 Then
    Fakulteta = 1
  Else
    ' rekurzivni klic zmanjša n,
    ' zato bo ta prej ali slej 1
    Fakulteta = n * Fakulteta(n - 1)
  End If
End Function
```

Rekurzivne klice lahko uporabljajo funkcije in procedure.

Klicanje podprogramov

Podprogram pokličemo tako, da zapišemo njegovo ime, ki mu sledijo dejanske vrednosti parametrov, naštete v enakem zaporedju, kakor so določene v glavi podprograma. Oglejmo si, kako je na splošno mogoče klicati podprograme. Za zgled naj nam bo naslednji podprogram:

```
Private Function Sporocilo(niz As String, vrsta As Integer) As Integer
  Sporocilo = MsgBox(niz, vrsta, "Sporocilo")
End Function
```

Ker v visual basicu lahko funkcijo kličemo kot proceduro, sta mogoča dva klica. Če funkcijo izkoristimo samo za izpis sporočila, jo pokličemo kot proceduro:

```
Sporocilo "Lep pozdrav", 0
```

Klic procedure je samostojen stavek, parametri pa so našteti na seznamu.

Po želji lahko isto funkcijo uporabimo tudi v izrazu in upoštevamo vrnjeno vrednost:

```
If Sporocilo("Nadaljujem?", vbQuestion + vbYesNo) = vbYes Then
  Sporocilo "Nadaljeval sem", 0
End If
```

Klic funkcije je del izraza, parametri pa so našteti na seznamu, ki ga *objemata* oklepaja. Pri klicu procedure oklepajev okrog seznama parametrov ne smemo zapisati. Naslednji zgled je napačen:

```
Sporocilo("Narobe!", 0)
```

Pritožil se bo že urejevalnik kode. Prav to, ali oklepaji so ali jih ni, prevajalniku pove, ali gre za samostojen stavek ali za del izraza.

NAMIG Pri podprogramih, ki sprejmejo samo en parameter, se zdi uporaba oklepajev dopustna tudi pri samostojnih stavkih. V tem primeru se parameter najprej izračuna in zatem posreduje. Če želimo, lahko oklepaje zapišemo okrog vsakega parametra posebej. Par oklepajev okrog *vseh* parametrov pa je na splošno dopusten le pri funkciji. To razliko nam kaže že urejevalnik kode:

```
Function Piskaj(n As Integer) As Boolean
Dim i
  For i = 1 To n
    Beep
  Next
  Piskaj = (n > 0)
End Function

Private Sub Command1_Click()
  If Not Piskaj(Text1.Text) Then Piskaj (1)
End Sub
```

Prvi klic podprograma Piskaj je funkcijski, zato se oklepaj stika z imenom funkcije. Drugi je proceduralni, zato je med imenom in parametrom presledek. Oklepaj tu zgolj povzroči, da se vrednost 1 najprej ovrednoti kot izraz in šele zatem posreduje proceduri. Kakor smo že omenili, podtikanje parametrov kot izrazov posredno onemogoči prenos po referenci.

V prejšnjih zgledih smo podprograme klicali neposredno. Tako lahko kličemo le podprograme, ki so zapisani v *istem* modulu, oziroma *javne* podprograme iz neodvisnih modulov s kodo (*code modules*). V projektih, pri katerih je več obrazcev, lahko kličemo tudi javne podprograme iz drugega obrazca. Omenili smo že, da so javni podprogrami obrazca pravzaprav njegove nove *metode*. Ustrezno temu jih tudi kličemo: ne neposredno, temveč prek sklicevanja na obrazec:

```
' Koda, ki pripada obrazcu z imenom Form1
Public Sub Command1_Click()
  MsgBox "Klik!"
End Sub

' Poskrbimo, da je na začetku viden tudi
' drugi obrazec
Private Sub Form_Load()
  Form2.Show
End Sub

' Koda, ki pripada obrazcu z imenom Form2
Private Sub Form_Click()
  Form1.Command1_Click
End Sub
```

Z ročnim popravkom pridevnika procedure Command1_Click iz Private v Public smo jo spremenili v *javno metodo* obrazca Form1, ki jo lahko sedaj pokličemo kjerkoli.

Posredno sklicevanje je včasih potrebno tudi za javne podprograme znotraj kodnih modulov. Ko je v dveh modulih določen javni podprogram z *enakim* imenom, je bolje natančno opredeliti, kateri podprogram želimo poklicati, drugače je izid lahko povsem nepričakovan:

```
Module1.Obdelaj x
Module2.Obdelaj x, 42
```

Vidimo, da je mogoče javne podprograme obravnavati tudi kot metode modula. Ime modula je razvidno iz projektnega okna. Po želji ga lahko spremenimo z oknom za nastavljanje lastnosti, saj tudi neodvisnim kodnim modulom pripada lastnost Name. Ker lahko isti modul vključimo v različne projekte, je tehniko posrednega sklicevanja najbolje vedno uporabljati.

Posebni parametri

Visual basic omogoča pisanje procedur, ki sprejmejo spremenljivo število parametrov. Zgled takega podprograma je MsgBox. Lahko mu predpišemo od enega do pet parametrov. Z drugimi besedami pravimo, da ima podprogram štiri *neobvezne parametre*.

Neobvezni parametri

Neobvezne parametre (*optional parameters*) določimo z rezervirano besedo **Optional** s seznama parametrov znotraj glave podprograma. Vsi parametri, ki morebiti sledijo neobveznemu, morajo biti tudi neobvezni. Oglejmo si zgled:

```
Private Sub Command1_Click()
  Izpis "A"
  Izpis "A", "B"
  Izpis "A", , "C"
  Izpis "A", "B", "C"
End Sub

Private Sub Izpis(p1, Optional p2, Optional p3)
  MsgBox CStr(p1)
  MsgBox CStr(p2)
  MsgBox CStr(p3)
End Sub
```

Naša lastna procedura izpiše vsebino treh parametrov. Zaradi posebne napovedi jo lahko pokličemo na različne načine. Poljubno število neobveznih parametrov je pri klicu moč izpustiti s seznama. Izpis, ki ga povzroči podprogram MsgBox z rabo funkcije CStr, nam razkrije, da izpuščeni parametri znotraj podprograma povzročijo napako s kodo 448. V sprotni pomoči lahko preberemo, da ta koda označuje – povsem pričakovano – manjkajoči parameter. Namesto da bi ugotavljali kodo napake, lahko uporabimo vgrajeno funkcijo IsMissing, ki vrne resnično vrednost za manjkajoče neobvezne parametre. Bolj trdoživa oblika procedure bi se lahko glasila takole:

```
Private Sub Izpis(p1, Optional p2, Optional p3)
  MsgBox p1
  If Not IsMissing(p2) Then MsgBox p2
  If Not IsMissing(p3) Then MsgBox p3
End Sub
```

Omenjena vrednost napake, ki ponazarja manjkajoči parameter, je lahko shranjena zgolj znotraj tipa Variant. Neobvezni parametri so lahko tudi osnovnih podatkovnih tipov (ne morejo biti uporabniško definiran tip). V tem primeru se manjkajočih parametrov ne da odkriti, saj ti dobijo privzeto vrednost, ki pripada posameznemu osnovnemu tipu:

```
Private Sub Command1_Click()
  Privzeto
End Sub

Private Sub Privzeto(Optional p1 As Integer, _
Optional p2 As Boolean, Optional p3 As String)
  MsgBox p1 ' 0
  MsgBox p2 ' False
  MsgBox p3 ' ""
End Sub
```

Če nam te privzete vrednosti ne ustrezajo, lahko neobveznim parametrom predpišemo kako drugo. Celotna oblika zapisa neobveznega parametra se namreč glasi:

Optional *imeParametra* [**As** *podatkovniTip*][= *privzetaVrednost*]

S tem si zagotovimo, da podprogram zadovoljivo opravi svojo nalogo tudi pri manjkajočih parametrih, npr. takole:

```
Private Sub Sporoci(Optional s As String = "(prazno sporocilo)")
  MsgBox s
End Sub
```

Omenimo še, da je tudi neobvezne parametre moč prenašati ali po vrednosti ali po referenci.

Poimenovani parametri

Veliko vgrajenih podprogramov lahko uporabimo tako, da jim posredujemo *poimenovane parametre* (*named parameters*). Ti nam olajšajo rabo podprogramov z veliko (ne)obveznimi parametri. Poimenovane parametre lahko zapišemo v poljubnem vrstnem redu, vendar moramo vsakega označiti z njegovim simboličnim imenom, kot je določen v glavi podprograma. Zato se nam izplača pisati podprograme z opisno poimenovanimi parametri. Tako definiran podprogram:

```
Private Sub Krava(strIme As String, Optional nStarost As Integer = -1)
  MsgBox strIme & ", starost: " & _
    IIf(nStarost = -1, "(neznana)", nStarost)
End Sub
```

lahko pokličemo z naslednjimi poimenovanimi parametri:

```
Krava strIme:="Liska"
Krava nStarost:=6, strIme:="Šeka"
```

Kakor vidimo, se poimenovani parameter zapiše najprej s simboličnim imenom, ki mu sledita dvopičje in enačaj (:=) ter dejanska vrednost. Vrstni red naštevanja poimenovanih parametrov zato ni pomemben. S poimenovanimi parametri se ne moremo izogniti podajanju *obveznih* parametrov, vendar njihov vrstni red ni pomemben.

> **NAMIG** V zgornjem zgledu smo uporabili posebno privzeto vrednost za ugotavljanje navzočnosti neobveznega parametra. Starost –1 namreč ni smiselna. Funkcija IIf ovrednoti prvi parameter, ki je logični izraz. Če je resničen, vrne vrednost drugega, sicer pa tretjega parametra, vedno pa ovrednoti oba!

Nedoločljivo število parametrov

Visual basic omogoča tudi določitev podprograma, ki sprejme *nedoločljivo* oziroma poljubno število parametrov. Parameter, ki ga označimo z rezervirano besedo **ParamArray**, je dinamično polje, vsebovano v prilagodljivem tipu Variant, ki vsebuje vse posredovane parametre.

```
Private Function Sestej(ParamArray n())
Dim i
  For i = LBound(n) To UBound(n)
    Sestej = Sestej + n(i)
  Next
End Function

Private Sub Command1_Click()
  MsgBox Sestej(1, 2, 3) + Sestej(4, 5, 6, 7, 8, 9)
End Sub
```

Parameter, označen s ParamArray, mora biti zadnji na seznamu. Njegove napovedi ne smemo kombinirati z drugimi pridevniki (ByVal, ByRef, Optional). Njegov tip je vedno Variant. Vsak tovrsten parameter je že po definiciji poljuben, saj je lahko posredovano polje tudi prazno.

Sklicevanja

Sklicevanje (*referencing*) je način dostopa do določenega elementa iz programske kode. Različne načine sklicevanja oziroma *naslavljanja*, kakor ga tudi imenujemo, spoznavamo praktično že od prvega zapisanega zgleda, tu pa si jih bomo natančneje ogledali.

Za vse elemente obrazca (zasebne in javne) lahko uporabimo *neposredno* sklicevanje. Neposredno sklicevanje zahteva samo zapis imena elementa. Nekaj zgledov:

```
Caption = "Obrazec"  ' Lastnost samega obrazca
Text1.Text = ""      ' Lastnost gradnika na obrazcu
Command1_Click       ' Zasebni podprogram obrazca
x = Preracuaj(1, 3)  ' Klic funkcije obrazca
```

Neposredno sklicevanje lahko uporabimo tudi za javne dele neodvisnih modulov s kodo. Za javne elemente drugih obrazcev ali pri reševanju imenskih konfliktov javnih modulov pa moramo poseči po *posrednem* sklicevanju, pri katerem najprej zapišemo ime obrazca oziroma modula. Nekaj zgledov:

```
Form2.Caption = "Obrazec2"   ' Lastnost drugega obrazca
Form2.Label1.Visible = False ' Lastnost gradnika drugega obrazca
x = Form2.Sestej(1, x, 45)   ' Klic zunanje javne funkcije
Module1.Koncaj               ' Klic zunanje javne procedure
```

Posredno sklicevanje lahko uporabimo vedno, tudi znotraj istega obrazca. S tem marsikdaj izboljšamo berljivost programske kode.

Pri pogostem sklicevanju na isti element nam pride prav stavek **With**. Z njim lahko programsko kodo precej skrajšamo. Naslednji zgled je enakovreden zadnjemu:

```
With Form2
  .Caption = "Obrazec2"      ' Lastnost drugega obrazca
  .Label1.Visible = False    ' Lastnost gradnika drugega obrazca
  x = .Sestej(1, x, 45)      ' Klic zunanje javne funkcije
End With
Module1.Koncaj               ' Klic zunanje javne procedure
```

Element, ki ga naslovimo v stavku With, je znotraj dosegljiv samo z zapisom operatorja sklicevanja (.). Stavke With lahko gnezdimo, vendar vedno velja zadnji. Kljub temu nam gnezdeni stavki včasih pridejo prav, npr. če je lastnost znova predmet:

```
With Form1
  .Height = 100     ' enako kot Form1.Height = 100.
  .Caption = "Lep pozdrav!"
  With .Font
    .Color = Red    ' enako kot Form1.Font.Color = Red.
    .Bold = True
  End With
End With
```

Nadzor izvajanja

Stavki za nadzor izvajanja so bistveni pri izgradnji zahtevnejših programov. Z njimi nadzorujemo tok izvajanja programske kode (*control flow*). Delimo jih na odločitvene in zančne. Prvi na podlagi ovrednotenega izraza cepijo tok izvajanja, drugi pa skupino stavkov ponavljajo.

Odločitveni stavki

Visual basic pozna dva odločitvena stavčna *bloka*: **If** in **Select Case**. Funkcionalno sta si podobna, vendar stavek If najpogosteje uporabljamo za eno- ali dvosmerne, Select Case pa za večsmerne cepitve toka izvajanja. Najpreprostejša oblika stavka If je *enovrstična*:

```
If pogoj Then stavek
```

Pogoj je najpogostejši logični izraz, lahko pa je tudi številski. Število nič se tolmači kot logična neresnica (*false*), nenična vrednost pa kot resnica (*true*). Če je pogoj resničen, se izvede en sam programski stavek, ki sledi. Omejitev na en stavek lahko presežemo tako, da z njim pokličemo podprogram ali pa zapišemo večvrstično obliko stavka If:

```
If pogoj Then
  stavki
  …
End If
```

Stavki, ki se izvedejo ob resničnem pogoju, se lahko nizajo do konca, ki ga določa stavek End If. Taki strukturi zato raje rečemo stavčni *blok* **If … End If**. S stavčnim blokom lahko zapišemo tudi večsmerne odločitve. Celotna oblika stavka v opisnem zapisu je:

```
If pogoj1 Then
  [stavki1]
[ElseIf pogoj2 Then
  [stavki2]]
  …
[Else
  [stavkiN]]
End If
```

V bloku lahko zapišemo poljubno število pogojev in stavkov, ki se izvedejo ob njihovi resnični vrednosti. Vse morebitne naslednje pogoje zapišemo s posebnim stavkom **ElseIf**. Stavčni blok, ki ga vpelje rezervirana beseda Else, se izvede, če nobeden od naštetih pogojev ni izpolnjen. Iz opisnega zapisa je razvidno, da sta stavka ElseIf in Else neobvezna in med seboj povsem neodvisna.

Nekaj zgledov:

```
If Not IsNumeric(Text1.Text) Then MsgBox "Vnesi število!"
```

```
If x < 0 Then
  x = -x
Else
  x = x * x
End If
```

```
If napaka = 1 Then
  niz = "napačen parameter"
ElseIf napaka = 2 Then
  niz = "deljenje z nič"
```

```
ElseIf napaka = 3 Then
  niz = "prekoračitev"
Else
  niz = "neznana napaka"
End If
```

V zadnjem zgledu bi lahko dodatne pogoje nizali še naprej, vendar je nenehno primerjanje istega izraza z različnimi konstantnimi vrednostmi naloga, ki jo pregledneje in učinkoviteje opravi stavčni blok Select Case. Blok **Select Case** na podlagi izida izraza določi točko nadaljevanja izvajanja kode. Njegova splošna oblika je:

```
Select Case izraz
  [Case seznamIzidov1
    [stavki1]]
  [Case seznamIzidov2
    [stavki2]]
  …
  [Case Else
    [stavkiN]]
End Select
```

Stavčni blok Select Case izraz ovrednoti le *enkrat*. Njegov izid je lahko poljubna konstantna vrednost, tudi niz znakov. Bloki Case, ki sledijo, določajo seznam konstantnih izidov, ki jim odgovarjajo pripadajoči programski stavki. Tudi tu imamo privzeto možnost, blok **Case Else**, ki se izvede, ko izida izraza ni mogoče najti na nobenem seznamu. Zadnji zgled, preoblikovan v blok Select Case, je videti takole:

```
Select Case napaka
  Case 1
    niz = "napačen parameter"
  Case 2
    niz = "deljenje z nič"
  Case 3
    niz = "prekoračitev"
  Case Else
    niz = "neznana napaka"
End Select
```

Oglejmo si še zgled, ko se pogojni izraz ovrednoti v niz, bloki Case pa vsebujejo sezname konstantnih vrednosti, zapisane na različne načine:

```
Select Case UCase(Mid(niz, i, 1))
  Case "Č", "Š", "Ž"
    MsgBox "Slovenska črka"
  Case "A" To "Z"
    MsgBox "Črka"
  Case Is > "Z"
    MsgBox "Ni črka"
End Select
```

Seznam je lahko poljubna kombinacija konstantnih vrednosti, *zaprtih* intervalov (zapišemo jih z rezervirano besedo **To**) in *polzaprtih* intervalov (zapišemo jih z

rezervirano besedo **Is** in primerjalnim operatorjem). Rezervirana beseda Is v blokih Case nadomesti *ovrednoteno* vrednost pogojnega izraza. Zgornji zgled deluje pri slovenskih krajevnih nastavitvah. Prepoznava vrsto znaka. Omeniti je treba, da se prvi in zadnji seznam prekrivata, vendar se po pravilih bloka Select Case vedno izvede *največ* en blok, ki ustreza pogoju. Takoj zatem se izvajanje prenese na stavke, ki sledijo End Select.

> **NAMIG** Blok Select Case ne more nadomestiti poljubnega bloka If ... ElseIf ... Else. Zamenjava je mogoča samo, ko se v bloku If nenehno vrednoti enak izraz. Zato je blok Select Case hitrejši, blok If pa prilagodljivejši.

Zanke

Za ponavljanje bloka programskih stavkov uporabljamo *zanke* (*loops*). Na splošno jih delimo na dve vrsti:

- Zanke, ki se izvajajo, dokler velja pogoj (stavčni blok **Do** ... **Loop**).
- Zanke z vnaprej predpisanim številom ponovitev (stavčni blok **For** ... **Next**).

Pri zankah **Do** zapišemo pogoj v obliki izraza, ki se ovrednoti bodisi v logično bodisi v številsko konstantno vrednost. Podobno kakor pri stavku If velja, da je logična neresnica (*false*) enakovredna nični, resnica (*true*) pa nenični številski vrednosti. Stavčni blok Do ... Loop lahko zapišemo v dveh oblikah. Pri prvi obliki:

```
Do {While|Until} pogoj
  stavki
Loop
```

se pogoj vrednoti *pred* vsako ponovitvijo zanke, zato je mogoče, da se stavki tudi nikoli ne izvedejo (0 ali več ponovitev zanke). Druga oblika:

```
Do
  stavki
Loop {While|Until} pogoj
```

zagotavlja vsaj eno ponovitev zanke, saj se pogoj vrednoti na *koncu* vsake ponovitve zanke (1 ali več ponovitev). Če ob pogoju zapišemo rezervirano besedo **While**, se zanka ponavlja, dokler je pogoj *resničen*, uporaba rezervirane besede **Until** pa – ravno nasprotno – ponavlja zanko, dokler je pogoj *neresničen*. Oglejmo si zgled:

```
Dim datum
Do
  datum = InputBox("Vnesi datum!")
Loop Until IsDate(datum)
MsgBox CVDate(datum)
```

Funkcija InputBox odpre pogovorno okno za vnašanje besedila. Zanka se ponavlja, dokler uporabnik ne vnese nečesa, kar je mogoče razumeti kot datum. Paziti moramo, da se pogoj znotraj zanke zagotovo kdaj spremeni. Če nismo previdni, lahko hitro zapišemo neskončno zanko.

> **NAMIG** Priporočljiva je uporaba rezervirane besede While, ki znatno olajša razumevanje kode. Po potrebi lahko vsak logični pogoj zanikamo z operatorjem Not, torej je Do Until povsem enakovredno Do While Not.
>
> Visual basic še vedno pozna tudi staro obliko zanke **While ... Wend**, katere raba ni več priporočljiva.

Zanke z vnaprej predpisanim številom ponovitev zapišemo s stavkom **For**. Namesto pogoja nastopa v teh zankah *števec*. Števec (*counter*) se v ponovitvah zanke sprehodi prek podanega intervala. Popolna oblika stavčnega bloka For ... Next se glasi:

```
For števec = začetekIntevala To konecIntervala [Step korak]
  stavki
Next [števec]
```

Števec je spremenljivka številskega tipa, katere vrednost je ob vstopu v zanko enaka predpisanemu začetku intervala. V vsaki ponovitvi zanke se števec spremeni za korak, zanka pa se izteče, ko števec preseže konec intervala. Korak je lahko pozitivna ali negativna vrednost, če ga izpustimo, se privzame vrednost 1. Če je konec intervala manjši od začetka (oziroma ravno nasprotno pri negativnem koraku), se zanka nikoli ne izvede. Zgled:

```
Dim n
For n = 30 To 1 Step -10
  MsgBox n ' 30, 20, 10
Next
```

Pri vključenem preverjanju napovedi spremenljivk moramo vnaprej napovedati tudi spremenljivko, ki rabi za števec zanke. Zanko For pogosto uporabljamo za sprehod prek vseh elementov polja:

```
Dim i
Dim a

a = Array("prvi", "drugi", "tretji")

For i = 0 To UBound(a)
  MsgBox a(i)
Next
```

Za pregledovanje polja imamo še boljšo in splošnejšo rešitev, posebno obliko zanke For, stavčni blok **For Each** ... **Next**. Blok je podoben običajni zanki For, le da mu ne predpišemo intervala, ampak skupino elementov, kot je polje podatkov, polje gradnikov ali *zbirka*. Namesto števca nastopa v zanki spremenljivka, ki v vsaki ponovitvi predstavlja drug element skupine. Splošna oblika zanke se glasi:

```
For Each element In skupina
  stavki
Next [element]
```

Stavek For Each je zelo uporaben pri sprehodih skozi *vse* elemente skupine. Vemo, da imajo polja podatkov lahko najrazličnejše intervale zaporednih števil. Še težje obvladljiva so polja gradnikov, saj imajo lahko povsem neodvisne nastavitve lastnosti Index, s poljubnimi vrzeli vmes. Zbirke so zelo pogosta podatkovna struktura v predmetnih ogrodjih (spoznali jih bomo v nadaljevanju). Naslednji zgled se sprehodi po vseh elementih polja:

```
Dim element
Dim polje

polje = Array("prvi", "drugi", "tretji")

For Each element In polje
  MsgBox element
Next
```

Vidimo, da elementov sploh ni treba neposredno naslavljati z zaporedno številko. Zančni blok poskrbi, da ima spremenljivka element v vsaki ponovitvi zanke novo vrednost. Tip spremenljivke mora ustrezati dejanski vsebini polja, zato lahko kot »števec« v zanki For Each uporabimo zgolj prilagodljivi tip Variant in ne katerega od osnovnih tipov. Naslednji zgled izpiše imena vseh gradnikov, izrisanih na obrazcu:

```
Dim gradnik

For Each gradnik In Form1.Controls
  MsgBox gradnik.Name
Next
```

Lastnost Controls je *zbirka* (*collection*), ki ponazarja vse gradnike na obrazcu. Spremenljivka gradnik se v vsaki ponovitvi zanke sklicuje na drug gradnik, zato lahko z njo pridemo do vseh lastnosti in metod gradnikov. Skupna lastnost vseh gradnikov je Name, ki jo v zanki izpisujemo. Če na obrazcu ni nobenega gradnika, se zanka ne izvede nikoli.

Če želimo, lahko izvajanje zanke prekinemo tudi predčasno. Ponavljanje zanke Do ... Loop prekinemo s stavkom **Exit Do**, zanke For ... Next pa s stavkom **Exit For**. Stavka se lahko v pripadajočih zankah znajdeta kjerkoli in se poljubno mnogokrat ponovita.

7. POGLAVJE

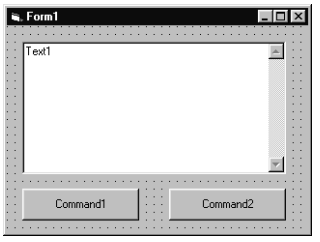
Predmeti in zbirke

Z visual basicom praktično ne moremo napisati programa, ki bi tako ali drugače ne uporabljal *predmetov*. Predmet (*object*) je ime za skupek podatkov in pripadajoče programske kode, ki se z njimi ukvarja. Osnovni predmeti, s katerimi se ukvarjamo že od začetka, so gradniki uporabniškega vmesnika: obrazec, gumb, seznam, menujska izbira... V programski kodi je na voljo še nekaj posebnih predmetov, ki jih med načrtovanjem ne moremo uporabiti. S tehnologijo predmetnega povezovanja nam visual basic omogoča celo uporabo predmetov drugih programov, ki so nameščeni v našem računalniku ali pa se izvajajo kje v omrežju, v katerega smo vključeni. In, ne nazadnje, nove predmete lahko v celoti izdelamo v samem visual basicu.

Razredi in izvodi

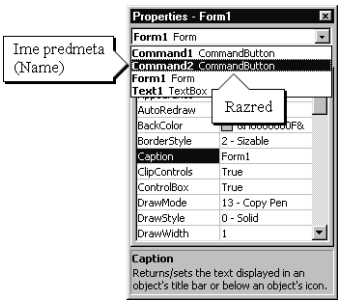
Za predmet smo že dejali, da je skupek podatkov in programske kode, tako kakor npr. obrazec uporabniškega vmesnika. Ne vemo pa še, da je vsak predmet dejansko *izvod* nekega *razreda*. Razred (*class*) je opis splošnega predmeta neke vrste in je za predmete to, kar je za navadne spremenljivke podatkovni tip. Izvod (*instance*) razreda je konkreten predmet z natančno opredeljenimi lastnostmi. Vsi gradniki, ki jih najdemo v orodjarni, so pravzaprav pripravljeni *razredi*. Gradnik, ki ga izrišemo na obrazcu, je *izvod* tega razreda oziroma *predmet*.

V oknu za nastavljanje lastnosti lahko za vsak gradnik ugotovimo, kateremu razredu pripada. Denimo, da smo izrisali uporabniški vmesnik, podoben tistemu na spodnji sliki:



Slika 7-1: Uporabniški vmesnik s tremi gradniki na obrazcu.

S seznama gradnikov okna za nastavljanje lastnosti lahko sedaj preberemo razred, ki mu posamezen gradnik pripada:



Slika 7-2: Seznam v oknu za nastavljanje lastnosti opiše razred predmeta.

Na obrazcu imamo dva gumba, ki sta oba predstavnika razreda CommandButton. Ker vsak predmet temelji na razredu, ima določeno *vedenje*, ki ga ponazarjajo njegove metode in dogodki. Vedenje je določeno za ves razred. Po drugi plati je vsak gumb *poseben* izvod razreda, z lastnim *stanjem*, ki ga opisujejo njegove lastnosti. Zato sta za začetek lastnosti Name in Caption med gumboma različni, spremenimo pa lahko še nekatere.

Predmetna urejenost

Če na uporabniški vmesnik izrišemo dva gumba, imata lahko različen napis, položaj in velikost, vendar sta oba izvoda razreda CommandButton. Delita pa si še dve skupni lastnosti:

- Oba sta na istem obrazcu.
- Oba sta pripadnika gradnikov (*controls*).

Za gradnika obrazec in okvir smo že dejali, da sta tudi vsebnika (*container*), torej lahko vsebujeta nove gradnike. Vsak gradnik (razen obrazca samega) je gotovo vsebovan v obrazcu. Zato so gradniki urejeni v hierarhično strukturo, ki je pomembna pri njihovem naslavljanju. V zgledih v tem delu smo že spoznali, da moramo vse javne elemente obrazca (javne spremenljivke, podprograme in tudi gradnike) naslavljati s sklicevanjem na sam obrazec. To je posledica njihove vsebovanosti.

Zbirke

Razredi, ki si delijo skupne lastnosti, so lahko vsebovani v *zbirki*. Zbirka (*collection*) je predmet z lastnimi lastnostmi in metodami, ki vplivajo na vse njene elemente

(*members*). Zbirka se v programski kodi vede kot posebna vrsta polja[23]. Vsak element zbirke je označen z zaporedno številko. Te se vedno začno z 0. Osnovne pripravljene zbirke v visual basicu so:

Ime zbirke	Opis
Forms	Zbirka vseh obrazcev projekta.
Controls	Zbirka vseh gradnikov na posameznem obrazcu.
Printers	Zbirka vseh nameščenih tiskalnikov.

Za ponazoritev si oglejmo programski zgled, ki preklaplja vidnost vseh gradnikov na obrazcu:

```
Private Sub Form_Click()
Dim c
  For Each c In Controls
    c.Visible = Not c.Visible
  Next
End Sub
```

Za sprehod po zbirkah često uporabimo zanko For Each. Vse zbirke poznajo lastnost Count, ki vsebuje skupno število njenih elementov, tako da je mogoče zanko zapisati tudi takole:

```
Private Sub Form_Click()
Dim i
  For i = 0 To Controls.Count - 1
    Controls(i).Visible = Not Controls(i).Visible
  Next
End Sub
```

Ker se zaporedne številke vedno začnejo z 0, je treba zančni števec ustaviti že pri Count-1.

Elemente zbirke je mogoče naslavljati tudi po njihovem imenu. Na voljo sta dva zapisa:

- Namesto zaporedne številke zapišemo kar ime predmeta kot konstantni niz znakov:

  ```
  Controls("Command1")
  ```

- Uporabimo posebni operator sklicevanja, ki ga označuje znak klicaj (!):

  ```
  Controls!Command1
  ```

Zbirke, predvsem njihov pomen in pripadnost večjim strukturam, so še posebej pomembne pri uporabi predmetnih knjižnic. Intenzivno jih bomo uporabljali pri

[23] To je posledica privzete lastnosti Item, ki je dejansko funkcijska metoda, ki vrne element zbirke s podano zaporedno številko.

dostopu do zbirk podatkov in pri nadzorovanju programov, ki podpirajo *avtomatizacijo*.

Vsebniki

Vsebovanost gradnika v kakem drugem podaja lastnost Container, ki jo poznajo vsi gradniki, razen samih vsebnikov. Ti so:

- Obrazec (gradnik *Form*).
- Okvir (gradnik *Frame*).
- Slikovni okvir (gradnik *PictureBox*).

Lastnost je dostopna le iz programske kode. Če imamo na obrazcu več vsebnikov, lahko lastnost Container za vsak posamezen gradnik tudi spremenimo. Ne moremo pa je uporabiti za spremembo obrazca, na katerem je gradnik.

Lastnost Container se od drugih razlikuje po tem, da ni vrednost katerega od osnovnih podatkovnih tipov, ampak nov predmet. Seveda, saj se sklicuje na gradnik – vsebnik. Naslednji zgled izpiše ime vsebnika ukaznega gumba:

```
Private Sub Command1_Click()
  MsgBox Command1.Container.Name ' Form1
End Sub
```

Ker gre za predmet, vsebnika ne moremo spremeniti z navadnim prireditvenim stavkom, ampak moramo uporabiti stavek Set. Če bi imeli na vmesniku tudi okvir z imenom Frame1, bi lahko gumb prestavili vanj:

```
Private Sub Command1_Click()
  Set Command1.Container = Frame1
End Sub
```

Preberimo si več o sklicevanju na predmet.

Predmetna spremenljivka

Podobno kakor lahko podatke osnovnih podatkovnih tipov shranjujemo v navadne spremenljivke, je mogoče predmete shranjevati v *predmetne* spremenljivke. Predmetna spremenljivka je simbolično ime za predmet nekega razreda.

Med navadnimi in predmetnimi spremenljivkami je pomembna razlika. Navadni spremenljivki že ob napovedi pripada zahtevani pomnilniški prostor, ki je nastavljen na privzeto vrednost za njen podatkovni tip. Predmetna spremenljivka lahko zgolj *kaže* na neki predmet. Njena začetna vrednost je neveljavna, zato jo moramo v praksi vedno naknadno določiti. To posledično pomeni, da lahko dve različni *predmetni* spremenljivki kažeta na *isti* predmet, medtem ko dve navadni

spremenljivki vedno predstavljata dve vsebini (ki sta po vrednosti sicer lahko enaki).

> **NAMIG** Predmetna spremenljivka v pomnilniku vedno zasede štiri zloge, ki vsebujejo naslov predmeta.

Napoved predmetne spremenljivke je podobna napovedi navadne:

```
{Dim|ReDim|Static|Private|Public} imeSpremenljivke[([seznamIntervalov])] _
As [New] razred
```

Predmetna spremenljivka je lahko vsebovana tudi znotraj prilagodljivega tipa Variant, zato bi lahko del As označili kot neobvezen, vendar po drugi strani spremenljivka tipa Variant ni značilna predmetna spremenljivka. Vidimo, da so tudi predmetne spremenljivke lahko združene v statična ali dinamična polja. Razred, ki je nekakšen podatkovni tip za predmetne spremenljivke, lahko podamo natančno (*specific*) ali splošno (*generic*). Imena specifičnih razredov lahko za vsak gradnik razberemo že iz namiga v orodjarni ali s seznama okna Properties. Splošni razredi so opisani v spodnji tabeli:

Razred	Opis
Object	Najbolj splošen tip, predstavlja lahko katerikoli razred.
Control	Katerikoli gradnik na obrazcu.
Form	Katerikoli obrazec.

> **NAMIG** Na voljo je še en splošni tip, MDIForm, ki predstavlja nadrejeni obrazec pri vmesniku z več odprtimi dokumenti. Praktično ga ne uporabljamo, saj ima lahko vsak projekt največ enega. Na voljo je le zaradi splošnosti, morda bo v prihodnosti mogoče razvijati tudi programe z več nadrejenimi obrazci.

Splošni podatkovni tipi so uporabni, ko ne vemo vnaprej, na kateri predmet se bo spremenljivka sklicevala, npr. če bi radi zapisali podprogram, ki prestavi poljuben gradnik:

```
Private Sub Diagonalno(c As Control, d As Integer)
  c.Move c.Left + d, c.Top + d
End Sub

Private Sub Command1_Click()
  Diagonalno Command1, 100
End Sub
```

Uporaba splošnih razredov je praktična, vendar je počasnejša in utegne biti nevarnejša. Če visual basicu natančno opišemo tip predmeta, bo lahko s popolnim prevajanjem (**Start With Full Compile**) zaznal sklicevanja na napačne lastnosti ali metode. Za povrh bo izdelal malenkost hitrejšo kodo. Oglejmo si še zgled, ki uporablja natančen tip:

```
Private Sub Command1_Click()
Dim vnos As TextBox
  Set vnos = Text1
  vnos.Text = "Besedilo..."
End Sub
```

Spremenljivka Vnos lahko kaže zgolj na vnosna polja (gradnik razreda TextBox). V zgornjem zgledu jo uporabimo kot sinonim za vnosno polje Text1.

Ustvarjanje in uničevanje predmetov

Vsaka novo napovedana predmetna spremenljivka ima na začetku neveljavno vrednost, ki jo ponazarja rezervirana beseda **Nothing**. Zato ji moramo pred praktično uporabo prirediti neki predmet, na katerega bo kazala. Za prirejanje vrednosti predmetni spremenljivki ne moremo uporabiti običajnega prireditvenega stavka, temveč moramo zapisati stavek **Set**[24]. V popolni obliki se glasi:

Set *predmetnaSpremenljivka* = {[**New**] *predmetniIzraz*|**Nothing**}

Kaj pomeni neobvezna rezervirana beseda **New**, ki smo jo spoznali že pri napovedi spremenljivke? Ker predmetna spremenljivka brez prirejene vrednosti ni uporabna, lahko določimo samodejno (implicitno) ustvarjanje novega izvoda, takoj ko se na predmetno spremenljivko sklicujemo iz programske kode. To dosežemo z rezervirano besedo New, ki jo zapišemo v *napovedi* spremenljivke:

```
Private Sub Command1_Click()
Dim frm As New Form1
  frm.Show
End Sub

Private Sub Command2_Click()
  Unload Me
End Sub
```

Zgornji zgled ponazarja ustvarjanje novih izvodov samega obrazca. Vidimo, da je Form1 pravzaprav ime *razreda*[25]! Spremenljivka frm je bila sedaj uporabna brez stavka Set, saj se ta implicitno zgodi ob prvem poseganju po metodi ali lastnosti

[24] Razlog je v spremenljivki Variant, ki se lahko prilagaja običajnim in predmetnim tipom. S stavkom Set lahko visual basic natančno ugotovi, kdaj želimo obravnavati predmet in kdaj običajno vrednost.

[25] Spomnimo se obravnave javnih delov obrazca. Omenili smo, da so javne spremenljivke prazaprav lastnosti, javni podprogrami pa metode obrazca.

predmeta, na katerega naj bi kazala (v našem primeru ob metodi Show). Implicitno ustvarjanje novega izvoda lahko dosežemo tudi v stavku Set. Spodnji zgled je enakovreden prejšnjemu:

```
Private Sub Command1_Click()
Dim frm As Form1
  Set frm = New Form1
  frm.Show
End Sub

Private Sub Command2_Click()
  Unload Me
End Sub
```

Raba rezervirane besede New v stavku Set je bolj priporočljiva in hitrejša.

> **NAMIG** V prejšnjih dveh zgledih se na nove izvode obrazca sklicujemo s krajevno spremenljivko, ki izgubi svojo vrednost ob izteku podprograma. Da bi se na novo ustvarjene predmete laže sklicevali, je bolje uporabiti statično ali javno spremenljivko. Pri obrazcih imamo vedno na voljo tudi zbirko Forms, ki v vsakem trenutku vsebuje vse obrazce projekta.

Implicitnega ustvarjanja novega izvoda z operatorjem New ne moremo uporabiti za nič drugega kakor za *lastne* razrede ali zbirke. Tak poizkus izdelave novega gradnika bi spodletel. Razen, kakor smo videli, pri obrazcih. Obrazci so nekaj posebnega, saj pravzaprav predstavljajo nove, uporabniško definirane razrede. Več o razvijanju lastnih razredov bomo izvedeli v nadaljevanju. Sedaj si oglejmo, kako lahko dinamično ustvarjamo in sproščamo tudi druge gradnike.

Dinamično ustvarjeni gradniki

Obrazec je v resnici nov, uporabniško določen razred. Drugi gradniki so vgrajeni in nespremenljivi, zato jih ne moremo ustvarjati neposredno z uporabo operatorja New. Njihovo dinamično dodajanje in odvzemanje je mogoče samo za *polje gradnikov* (*control array*). Ustvarjanje polja gradnikov smo si ogledali, ko smo obravnavali lastnost Index (glej opis gumbov, stikal in okvirov).

Nov element polja gradnikov dodamo s stavkom Load, odstranimo pa ga s stavkom Unload. Za preizkus naslednjega zgleda moramo na uporabniški vmesnik namestiti ukazni gumb z imenom Command1 in mu nastaviti lastnost Index na vrednost 0. S tem ga uvrstimo v polje gradnikov, ki mu lahko dinamično dodajamo in odvzemamo elemente:

```
Private Sub Command1_Click(Index As Integer)
Static i As Integer
  If Index = 0 Then
    i = i + 1
```

```
    Load Command1(i)
    Command1(i).Top = Command1(i).Top + i * (Command1(0).Height)
    Command1(i).Visible = True
  Else
    Unload Command1(Index)
  End If
End Sub
```

Polje gradnikov mora vsebovati vsaj en element (v posebnih primerih je lahko neviden). Zato v zgornjem zgledu odstranjujemo le elemente, katerih zaporedna številka ni nič. Vsak dinamično ustvarjen gradnik je na istem mestu kakor njegov predhodnik in za povrh še neviden. Zato ga prestavimo in mu popravimo lastnosti Visible. Spremenljivka i mora vedno vsebovati še neuporabljeno zaporedno številko, zato je napovedana kot statična.

NAMIG Visual basic za vsak obrazec projekta *samodejno* napove pripadajočo javno predmetno spremenljivko z implicitnim ustvarjanjem. Njeno ime je enako imenu obrazca, kakor bi v del splošnih napovedi zapisali:

```
Public Form1 As New Form1
```

Med načrtovanjem zato lahko pripravljene obrazce prikazujemo zgolj z metodo Show in jih v praksi nikoli ne nalagamo s stavkom Load.

Tudi obrazec lahko odstranimo s stavkom Unload. Pravzaprav je to edino pravilno odstranjevanje obrazca. Če tako odstranimo izhodiščni obrazec projekta, končamo izvajanje programa (razen če tega ne prekliče odzivna procedura dogodkov QueryUnload ali Unload). Končevanja izvajanja programa s samostojnim stavkom **End** zato ne priporočajo več, saj lahko določene razrede ohrani v pomnilniku brez veljavnega kazalca nanj in tako zmanjšuje količino prostega pomnilnika. Pravilen konec dela, enakovreden izbiri ukaza Close iz nadzornega menija okna, dosežemo s stavkom:

```
Private Sub Command1_Click()
  ' pravilno zapiranje obrazca in posledično tudi programa
  Unload Me
End Sub
```

NAMIG Če v razvojnem okolju visual basica izberemo ukaz End iz menuja Run ali kliknemo pripadajoči gumb v vrstici gumbov, prav tako povzročimo nepravilen konec programa, kakor bi poklicali stavek End. V programih, ki ne izdelujejo novih izvodov razredov, to nima hujših posledic. Te težave se moramo zavedati, kadar ustvarjamo nove izvode lastnih razredov.

Uničenje predmeta

Predmetna spremenljivka na začetku kaže na neveljaven predmet, ponazorjen z rezervirano besedo Nothing. Pozneje ji lahko s stavkom Set prirejamo različne predmete, tudi nove izvode razredov, ustvarjenih z operatorjem New. Kako dolgo živijo ti novi izvodi?

Vsak novo ustvarjeni izvod razreda živi, dokler se nanj še sklicuje kaka predmetna spremenljivka. Sklicevanje prekinemo tako, da spremenljivki eksplicitno priredimo vrednost Nothing:

```
Dim predmet As Object

Set predmet = New MojRazred

' delo s predmetom...

Set predmet = Nothing
```

Zgornji zgled uporablja poseben razred, tako imenovani MojRazred, ki ga določimo v posebnem, razrednem modulu. Tega bomo spoznali v nadaljevanju. Če bi na novi predmet kazala še kaka spremenljivka, bi morali izničiti tudi njeno sklicevanje. Šele tedaj bi predmet dejansko izginil in sprostil pomnilnik, ki mu pripada. Pri krajevnih predmetnih spremenljivkah, ki jih uporabljamo samo znotraj podprograma, eksplicitna prekinitev sklicevanja včasih niti ni potrebna. Če spremenljivka ni statična, umre ob koncu izvajanja podprograma in s tem sprosti tudi morebitna sklicevanja.

Izjema za tovrstno ukinjanje predmetov je obrazec. Čeprav spremenljivko, ki se sklicuje na nov, dinamično ustvarjen obrazec, nastavimo na vrednost Nothing, s tem še nismo izčrpali vseh sklicevanj nanj. *Vsakemu* obrazcu namreč pripada še kazalec iz zbirke Forms. Obrazec zato pravilno odstranimo le z ustreznim stavkom Unload.

Predmetni izrazi

Pri delu s predmeti lahko uporabimo nekaj posebnih operatorjev in rezerviranih besed. Pomen operatorja New, ki ustvari nov izvod razreda, smo natančno spoznali pri opisu ustvarjanja in uničevanja predmetov. Drug, zelo pomemben operator, je primerjalni operator **Is**.

Podobno kakor predmetnim spremenljivkam ne moremo prirejati novih vrednosti z običajnim prireditvenim stavkom, jih tudi ne moremo primerjati kar z enačajem. Enakost dveh predmetnih spremenljivk moramo najprej definirati: dve predmetni spremenljivki sta enaki, če kažeta na *isti* predmet. Operator Is v tem primeru vrne resnično vrednost:

```
Private Sub Command1_Click()
Dim p1 As Form1
Dim p2

  Set p1 = Me
  Set p2 = p1
  If p1 Is p2 Then MsgBox "Enaka sta!"

End Sub
```

Zgornji zgled nam ponazarja, da se tudi spremenljivka prilagodljivega tipa Variant lahko sklicuje na predmet.

Pri delu s predmeti je zelo pomemben operator **TypeOf**, ki preverja pripadnost predmeta določenemu razredu. Njegovo uporabo ponazarja spodnji zgled:

```
Private Sub Command1_Click()
Dim p1

  Set p1 = Me
  If TypeOf p1 Is Form Then MsgBox "Nek obrazec!"
  If TypeOf p1 Is Form1 Then MsgBox "Obrazec Form1!"

End Sub
```

Pripadnost razredu je resnična tudi, če natančno določen razred sodi v skupino splošnih razredov. V zgornjem zgledu sta resnična oba izraza, saj sodi Form1 med splošne obrazce. Operatorja Is in TypeOf vračata logično vrednost, zato lahko njun izid vpletemo v poljuben logični izraz. Po prioriteti sta zadnja med primerjalnimi operatorji, a še vedno pred vsemi logičnimi.

Pri sklicevanju na predmete imamo na voljo tudi dve posebni rezervirani besedi:

- **Nothing** – rezervirana beseda, ki predstavlja neveljaven predmet.
- **Me** – rezervirana beseda, ki predstavlja trenutni izvod predmeta.

Rezervirana beseda Me je dostopna v vseh *razrednih* procedurah. Edine razredne procedure, s katerimi smo se ukvarjali doslej, so procedure znotraj obrazca, za katerega že vemo, da je poseben razred. V tem primeru lahko Me uporabimo kot sinonim za izvod obrazca, s katerim se koda trenutno ukvarja. Za ponazoritev se lahko vrnemo k zgledu dinamičnega ustvarjanja obrazcev (ob razlagi operatorja New). Tam smo zapisali tudi odzivno proceduro, ki je obrazec odstranila:

```
Private Sub Command2_Click()
  Unload Me
End Sub
```

Klik ukaznega gumba bo vedno odstranil natančno tisti izvod obrazca, ki mu sam pripada.

DRUGI DEL

Delavnica

V TEM DELU

Okenski komandant
Urejevalnik
Risar

Drugi del knjige obsega tri zaokrožene zglede, ki so uporabni programi visual basica. Ob njih bomo spoznavali značilne težave pri izgradnji programa in kako jih odpraviti. To bo hkrati tudi priložnost za obravnavo nekaterih novih gradnikov. Denimo nekaterih vgrajenih, ki smo jih v prvem delu izpustili (npr. gradniki za dostop do datotečnega sistema), predvsem pa zunanjih, ki marsikatero sicer zapleteno opravilo poenostavijo na nastavljanje nekaj lastnosti in klic kake metode.

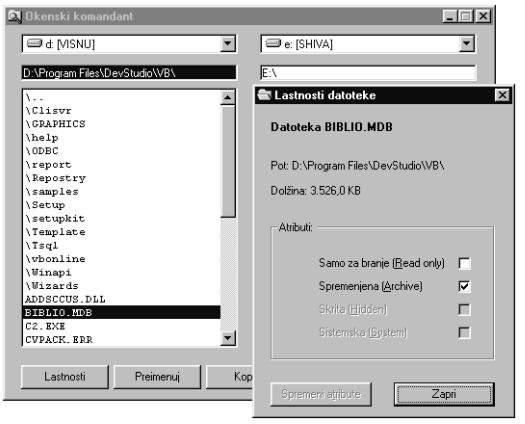
8. POGLAVJE

Okenski komandant

Prvi zaokroženi zgled, ki ga bomo razvili skupaj, bo program Okenski komandant. Že po imenu lahko sklepamo, s čim se bo ukvarjal. Naštejmo programerske tehnike, ki jih bomo spoznali pri njegovem razvijanju:

- Uporaba gradnikov za dostop do datotečnega sistema.
- Funkcije za delo z nizi.
- Nizkonivojsko delo z miško in tipkovnico.
- Posnemanje pritiskanja na tipke.
- Shranjevanje nastavitev v register.
- Uporaba vgrajenega razhroščevalnika in predmeta Debug.
- Upravljanje datotek.
- Delo z več obrazci.
- Oblikovanje izpisa.

Videz delujočega programa si lahko ogledate na spodnji sliki. Izvorna koda zgledov je na priloženi plošči CD-ROM, v mapah znotraj mape \Primeri\2-Delavnica\1-Komandant\.

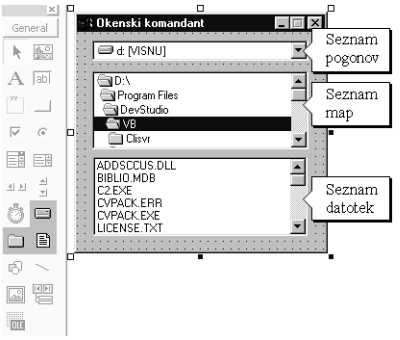


Slika 8-1: Okenski komandant v rabi

Seznami pogonov, map in datotek

Med vgrajene gradnike (*intrinsic controls*), večino smo jih izčrpno spoznali v prvem delu, sodijo tudi tri posebne oblike seznamov, ki prikazujejo različne elemente datotečnega sistema (*file system*). Z Okni 95 se je spremenilo izrazoslovje, ki ga opisuje. Imeniki (*directories*) so postali *mape* (*folders*). Prav tako se je spremenila navidezna hierarhija pogonov (*drives*) in map, saj se vse začenja na *namizju* (*desktop*), ki je posebna mapa. Kljub tem spremembam 32-bitni visual basic še vedno ponuja enake vgrajene gradnike kakor prejšnje 16-bitne različice. To je po svoje razumljivo, saj je za programerja izhodišče datotečnega sistema še vedno pogon, namizje pa je le ena izmed map. Po drugi strani pa ni redko, da v uporabniškem vmesniku potrebujemo npr. samo seznam razpoložljivih pogonov. S tremi posebnimi seznami lahko zgradimo obrazce po lastnih željah in potrebah, za prikaz pogovornih oken, ki so enakovredna sistemskim, pa imamo na voljo poseben gradnik ActiveX (spoznali ga bomo v naslednjem poglavju).

Preden se lotimo obravnave lastnosti in dogodkov teh posebnih seznamov, si pripravimo uporabniški vmesnik, podoben tistemu na spodnji sliki:



Slika 8-2: Seznami za prikaz datotečnega sistema

Gradniki med načrtovanjem programa prikazujejo nespremenljivo vsebino, ki je odsev *dejavne poti*. Če bi tako pripravljen vmesnik pognali brez ene same vrstice kode, bi ugotovili, da se vsak gradnik zase sicer vede pravilno, med seboj pa niso prav nič usklajeni. Uskladitev lahko dosežemo le z ustrezno programsko kodo. Pred tem si oglejmo še lastnosti:

Lastnosti

Lastnost **Drive**

Lastnost je na voljo samo pri seznamu pogonov (*DriveListBox*) in opisuje izbrani pogon. Vrednost, ki jo vrne, je niz z daljšim opisom pogona, npr:

Pogon	Lastnost Drive
Disketni pogon	"a:"
Drugi vgrajeni pogoni	"c: [*oznaka*]"
Omrežni pogoni	"x: *strežnik**deljenaPot*"

Pri nastavljanju lastnosti je pomemben le prvi znak niza, ki mora predstavljati veljaven pogon. Vsaka sprememba lastnosti Drive povzroči dogodek Change. Če vrednost nastavimo na neveljaven znak (neobstoječ ali nepripravljen pogon), povzročimo napako med izvajanjem.

Lastnost **Path**

Datotečna *pot* (*path*) je natančen opis kraja datoteke oziroma mape v datotečnem sistemu. Vsebovana je v lastnosti Path, ki je na voljo pri seznamu map (*DirListBox*) in datotek (*FileListBox*). Pot lahko podamo v krajevni obliki (relativno ali absolutno) ali v omrežni obliki UNC:

Pot	Zapis	Predstavlja
`.`	Relativen.	Trenutno dejavna mapa v dejavnem pogonu (dejavna pot).
`..`	Relativen.	Predhodna mapa v dejavnem pogonu.
`\zgledi\vaje`	Relativen	Podmapa `vaje`, vsebovana v osnovni mapi `zgledi` v dejavnem pogonu.
`d:\temp`	Absoluten	Osnovna mapa `temp` v pogonu `d:`. Absolutno pot sestavljata pogon in popoln opis poti do izbrane mape.
`\\Herkul\disk_c\podatki`	UNC	Mapa `podatki` na deljeni poti `disk_c` strežnika z imenom `Herkul`.

NAMIG Programi, ki želijo biti združljivi z 32-bitnimi Okni 95 in NT, morajo znati uporabljati imena datotek UNC (*Universal Naming Convention*). V tej obliki je mogoče preprosto nasloviti datoteko v kateremkoli računalniku v omrežju. Gradniki, s katerimi se ukvarjamo, ne omogočajo neposrednega dostopa do deljenih poti strežnikov. Za brskanje po omrežju je najenostavneje uporabiti standardna sistemska pogovorna okna, ki jih v visual basicu lahko uporabimo s posebnim gradnikom ActiveX. Spoznali ga bomo v naslednjem poglavju.

Če lastnost nastavimo na novo vrednost, povzročimo dogodek Change (pri seznamu pogonov) oziroma PathChange (pri seznamu datotek). Nova vrednost se pozna na vsebini gradnikov, vendar ne spremeni *dejavne poti*. Dejavna pot (*default path*) je absolutna pot, ki opisuje dejavni pogon in v njem izbrano mapo. Dejavno pot lahko preberemo s funkcijo **CurDir**, spremenimo pa s procedurama **ChDir** in **ChDrive**. Zgled:

```
MsgBox CurDir ' npr. c:\windows
Dir1.Path = "d:\temp"
MsgBox CurDir ' še vedno c:\windows
ChDir "d:\temp"
MsgBox CurDir ' še vedno c:\windows
ChDrive "d, to je pogon, ki ga želim"
MsgBox CurDir ' končno d:\temp
```

Za usklajenost dejavne in prikazane poti na seznamu map vedno potrebujemo tri programske stavke (spremembi lastnosti Path morata slediti še stavka ChDrive in ChDir). Stavek ChDir spremeni pot v poljubnem pogonu, vendar ne spremeni dejavnega pogona. V zgornjem zgledu smo najprej določili novo pot za pogon `d:`. Ker je bil dejavni pogon še vedno `c:`, je celotna dejavna pot ostala na `c:\windows`. Šele stavek ChDrive, pri katerem se iz podanega niza upošteva le prvi znak, je dejavno pot prestavil na `d:\temp`.

Tovrstna usklajenost pa večinoma sploh ni potrebna. Dejavno in izbrano pot uskladimo kvečjemu pred obsežnejšimi posegi v izbrani mapi, opisanimi z relativnimi potmi. Če ves čas uporabljamo zgolj absolutne poti, nas dejavna pot pravzaprav niti ne zanima.

Lastnosti **Pattern**, **ReadOnly**, **Archive**, **Normal**, **System** in **Hidden**

Naštete lastnosti pozna le seznam datotek. Z njimi lahko natančneje določimo vsebino seznama. Na seznamu bodo le datoteke, katerih imena ustrezajo vzorcu, določenem z lastnostjo Pattern. V vzorcu lahko uporabimo posebne znake `*`, `?` in `;`. Nekaj značilnih vzorcev je naštetih v spodnji tabeli:

Vzorec	Ustreza imenu
`*.*`	Ustreza vsem imenom (privzeta nastavitev).
`*.txt`	Imena s podaljškom .TXT
`a*;*.a*`	Imena ali podaljški, ki se začnejo z A.
`???.??t`	Imena, dolga natančno tri znake, katerih podaljšek ima tri znake in se konča s T.

Skratka, zvezdica (`*`) nadomesti poljubno (tudi prazno) zaporedje, vprašaj (`?`) poljuben znak, podpičje (`;`) pa združi več vzorcev.

Vsaki datoteki pripada še nekaj atributov, ki jo natančneje opisujejo. Ponazorjeni so z naslednjimi lastnostmi:

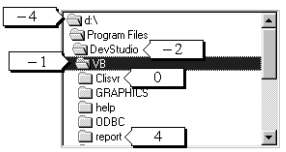
Lastnost	Atribut
Normal	Navadne datoteke (privzeta nastavitev True).
ReadOnly	Datoteka je na voljo samo za branje (privzeto True).
Archive	Datoteka se je spremenila od zadnjega arhiviranja (privzeto True).
Hidden	Skrita datoteka (privzeto False).
System	Sistemska datoteka (privzeto False).

Vsako izmed naštetih lastnosti lahko nastavimo na resnično ali neresnično vrednost. Prikazane datoteke imajo kvečjemu tiste atribute, katerega pripadajoča lastnost je nastavljena na resnično vrednost. Po privzetih nastavitvah torej niso vidne skrite in sistemske datoteke.

Lastnost **ListIndex**

Lastnost ListIndex pozna vsak seznam. Opisuje zaporedno številko izbrane vrstice, prvi pripada 0, če ni izbrana nobena, pa ima lastnost vrednost –1. Lastnost se vede nekoliko drugače le pri seznamu map (*DirListBox*). Tam vrednost –1 opisuje trenutno izbrano mapo, njej nadrejene mape imajo manjše vrednosti, prva njej podrejena mapa pa ima zaporedno številko 0. To prikazuje spodnja slika:

Slika 8-3: Lastnost ListIndex za seznam map



Zato nam pri sprehodu prek vseh elementov seznama lastnost ListCount bolj malo pomaga. Na srečo se posebej lepo vede lastnost List. Ta je vedno definirana za ves veljavni razpon zaporednih števil, le da pri neveljavnih vrsticah vrača prazen niz, pri veljavnih pa vrne *celotno* pot v absolutni obliki. Spodaj sta za zgled zapisana dva sprehoda, prvi izpiše in prešteje vse nadrejene, drugi pa vse podrejene mape:

```
' Sprehod po nadrejenih mapah
Dim intNadrejenih As Integer
intNadrejenih = -2
Do Until Dir1.List(intNadrejenih) = ""
  MsgBox Dir1.List(intNadrejenih)
  intNadrejenih = intNadrejenih - 1
Loop
intNadrejenih = Abs(intNadrejenih + 2)
MsgBox "Skupaj nadrejenih map: " & intNadrejenih

' Sprehod po podrejenih mapah
Dim intPodrejenih As Integer
intPodrejenih = 0
Do Until Dir1.List(intPodrejenih) = ""
```

```
  MsgBox Dir1.List(intPodrejenih)
  intPodrejenih = intPodrejenih + 1
Loop
MsgBox "Skupaj podrejenih map: " & intPodrejenih
```

Za nadrejene mape se zaporedno število vrstice zmanjšuje od –2 pa do neveljavne vrednosti, ki jo ponazarja prazen niz. Celotno število je torej za dva manjše od absolutne vrednosti zaporedne številke zadnje nadrejene mape. Pri podrejenih mapah je stvar preprostejša, zaporedna števila si sledijo podobno kakor pri drugih seznamih.

Dogodki

Gradniki sami skrbijo za prikaz vsebine, vendar jih moramo med seboj ročno uskladiti. Pomaga nam dogodek Change. Seznam datotek pozna še dva zanimiva dogodka, ki ju uporabimo ob posebnih priložnostih.

Dogodek **Change**

Dogodek je ključ do medsebojne uskladitve vsebine gradnikov. Pripeti se ob vsaki spremembi poti, bodisi programski (sprememba lastnosti) bodisi uporabniški (izbira s seznama). Odzivno proceduro moramo zapisati za seznam pogonov in seznam datotek. Njun videz je:

```
Private Sub Dir1_Change()
  File1.Path = Dir1.Path
End Sub

Private Sub Drive1_Change()
  Dir1.Path = Drive1.Drive
End Sub
```

Dogodka **PathChange** in **PatternChange**

To, kar je dogodek Change za seznam pogonov in map, je dogodek PathChange za seznam datotek. Za povrh pozna še dogodek PatternChange, ki se pripeti ob vsaki spremembi njegove lastnosti Pattern. Odzivno proceduro tega dogodka zapišemo, če bi po spremembi vzorca radi osvežili še kak drug gradnik (npr. oznako, ki prešteje vse datoteke na seznamu):

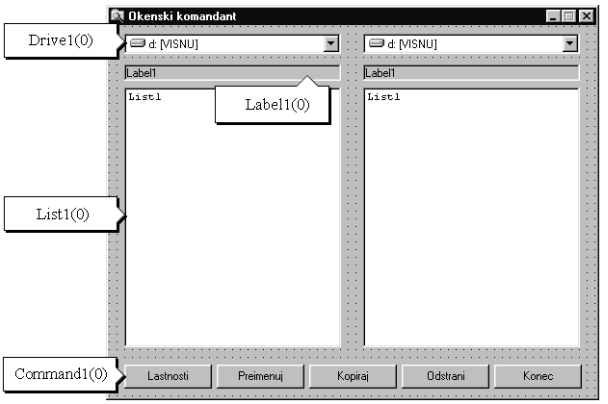
```
Private Sub File1_PatternChange()
  Label1.Caption = "Skupaj " & File1.ListCount & " datotek"
End Sub
```

NAMIG Celotno izvorno kodo, ki smo jo uporabili pri opisu gradnikov, najdete na priloženi plošči CD-ROM v mapi \Primeri\2-Delavnica\1-Komandant\1-Gradniki\.

Uporabniški vmesnik

Trije seznami za prikaz datotečnega sistema nam omogočajo veliko prilagodljivost pri načrtovanju uporabniškega vmesnika. Z vmesnikom za »Okenski komandant« želimo čim bolj posnemati videz našega vzornika, znamenitega programa Norton Commander. Tu pa naletimo na oviro. Želimo si, da bi naš vmesnik na *istem* seznamu prikazoval mape in datoteke znotraj dejavne poti. Takega gradnika, žal, ne poznamo. Rešitev je v tem, da uporabimo navaden odprti seznam (*ListBox*), ki ga polnimo s programsko kodo. Od vseh gradnikov za neposredno delo z datotečnim sistemom bomo tako ohranili zgolj seznam pogonov.

Pripraviti si moramo uporabniški vmesnik po zgledu spodnje slike:



Slika 8-4: Uporabniški vmesnik Okenskega komandanta

Vmesnik je simetričen, vsak gradnik na obrazcu je element polja gradnikov (*control array*). Gradniki na levi strani imajo zaporedno številko, določeno z lastnostjo Index, enako 0, desni strani pa pripada številka 1. Gumbi v spodnji vrstici pripadajo polju gradnikov z zaporednimi števili od 0 do 5. Taka organiziranost gradnikov nam bo izdatno olajšala kodiranje. Povzetek lastnosti, ki jih nastavimo med oblikovanjem, je v spodnji tabeli:

Gradnik	Lastnost	Vrednost
obrazec (*Form*)		
Form1	BorderStyle	1 - Fixed Single
	Caption	"Okenski komandant"
	Icon	(datoteka Form1.ico)[26]

[26] Datoteke binarnih sredstev lahko najdete na priloženi plošči, v mapi \Primeri\2-Delavnica\1-Komandant\2-Vmesnik. Ikone so povzete iz visual basicu priložene zbirke grafičnih elementov, le da so preimenovane v ime gradnika, ki mu pripadajo. Ikona v ničemer ne vpliva na preostanek zgleda, zato lahko izberete tudi katero koli drugo.

Gradnik	Lastnost	Vrednost
	KeyPreview	True
	MinButton	True
	StartUpPosition	2 - CenterScreen
oznaki (*Label*)		
Label1	Index	0
	BorderStyle	1 - Fixed Single
Label1	Index	1
	(drugo enako kakor pri gradniku z zaporedno številko 0)	
seznama pogonov (*DriveListBox*)		
Drive1	Index	0
Drive1	Index	1
seznama (*ListBox*)		
List1	Index	0
	Font	(Courier New, velikost 8)
	MultiSelect	1 – Simple
	Sorted	True
List1	Index	0
	(drugo enako kakor pri gradniku z zaporedno številko 0)	
gumbi (*CommandButton*)		
Command1	Index	0
	Caption	"Lastnosti"
Command1	Index	1
	Caption	"Preimenuj"
Command1	Index	2
	Caption	"Kopiraj"
Command1	Index	3
	Caption	"Odstrani"
Command1	Index	4
	Caption	"Konec"

NAMIG Vmesnik bomo najlaže oblikovali tako, da najprej določimo levo stran, nato pa gradnike z rabo odložišča podvojimo in prestavimo še desno. Na koncu dodamo še ukazne gumbe pri dnu. Uporabniški vmesnik Okenskega komandanta, razvit do te točke, najdete tudi na priloženi plošči CD-ROM, v mapi: \Primeri\2-Delavnica\1-Komandant\2-Vmesnik.

Programska koda

Po risanju vmesnika in nastavljanju lastnosti pride na vrsto pisanje programske kode. Zaradi boljše preglednosti jo bomo razvijali po korakih. Prvi problem, ki ga moramo rešiti, je polnjenje seznama z datotekami in mapami na izbrani poti.

Sprehod po mapi

Za ugotavljanje, ali je določena datoteka v mapi, imamo na voljo funkcijo **Dir**. Če si ogledamo njen opis v sprotni pomoči, ugotovimo, da se zna funkcija sprehoditi prek vseh datotek izbrane mape. Opisni zapis klica funkcije se glasi:

```
Dir[(pot[, atributi])]
```

Čeprav sta oba parametra neobvezna, moramo ob prvem klicu funkcije podati vsaj pot, drugače pride do napake med izvajanjem. Vrnjena vrednost je ime datoteke na določeni poti. Če podane poti ni ali tam ni moč najti nobene datoteke, funkcija vrne prazen niz. Drugače pa jo lahko pokličemo znova, *brez* parametrov, in vrnila bo ime naslednje datoteke na določeni poti, dokler je še kaka. Pot ima lahko na koncu vzorec, ki mu morajo vrnjena imena ustrezati. Podobno kakor pri lastnosti Pattern za seznam datotek lahko uporabimo znaka `*` in `?` (če vzorca ne podamo, se privzame `*.*`). Vrstni red vračanja datotek ni posebej določen. Z atributi lahko dodatno opredelimo, katere datoteke želimo najti. Poleg atributov, ki smo jih omenili že pri obravnavanju seznama datotek, imamo na voljo še dve posebni vrednosti, ki ju ponazarjajo naslednje vgrajene konstante:

Konstanta	Opis
vbNormal	Navadne datoteke.
vbHidden	Skrite datoteke.
vbSystem	Sistemske datoteke.
vbVolume	Oznaka pogona (če podamo ta atribut, se drugi zanemarijo).
vbDirectory	Mape.

Zadnja konstanta je še posebej zanimiva. Očitno je, da nam funkcija Dir lahko vrača tudi imena map na pregledovani poti.

Za sprehod bomo zapisali lasten podprogram. Pisanje novega podprograma je smiselno za vse kose kode, ki jo bomo klicali z več mest. Ker moramo pri Okenskem komandantu določiti vsebino levega in desnega seznama, ki lahko kažeta na različno pot, je najbolje zapisati proceduro, ki sprejme dva parametra: pot in zaporedno številko seznama. Preden se lotimo dela, napovejmo še dve modulski

spremenljivki, ki nam bosta pomagali pri delu. V del splošnih napovedi (*general declarations*) zapišemo:

```
Private strPot(1) As String
```

Spremenljivka `strPot` je polje *dveh* nizov (ne pozabimo, da je po privzeti nastavitvi spodnja meja 0). Spremenljivko smo poimenovali s predpono `str`, ki ponazarja njen tip. Sedaj se lahko lotimo procedure `PreglejPot`:

```
Private Sub PreglejPot(strNovaPot As String, intStran As Integer)
Dim strDatoteka As String

  ' preverimo, ali pot kaže na mapo:
  If Right(strNovaPot, 1) <> "\" Then strNovaPot = strNovaPot & "\"

  strPot(intStran) = strNovaPot
  Label1(intStran).Caption = strPot(intStran)
  List1(intStran).Clear

  ' Najdemo morebitno prvo datoteko, zahtevamo tudi mape
  strDatoteka = Dir(strPot(intStran) + "*.*", vbDirectory)

  ' Ponavljamo, dokler imamo kako datoteko
  Do While strDatoteka <> ""

    ' Zanemarimo trenutno mapo, ki jo ponazarja .
    If strDatoteka <> "." Then
      ' Ali gre za mapo?
      If (GetAttr(strPot(intStran) & strDatoteka) And vbDirectory) = _
        vbDirectory Then
        ' Mape predstavimo z začetnim \
        List1(intStran).AddItem "\" & strDatoteka
      Else
        ' Druge datoteke uvrstimo neposredno
        List1(intStran).AddItem strDatoteka
      End If
    End If

    ' Zahtevamo naslednjo datoteko
    strDatoteka = Dir

  Loop

End Sub
```

V proceduri najprej preverimo parameter nove poti. Če se ne končuje s pravim znakom, ga dodamo. Zatem shranimo novo pot v modulsko spremenljivko, tako da bo na voljo vsem podprogramom. Sledi obnavljanje vsebine gradnikov. Najprej osvežimo pot v oznaki, zatem pobrišemo morebitno vsebino seznama, na koncu pa v zanki zapolnimo seznam z novo vsebino. Prvi klic funkcije Dir zahteva, da nam funkcija vrača imena vseh datotek, tudi map[27]. Masko `*.*` smo na koncu dodali le zaradi boljše berljivosti kode, saj je tako ali tako privzeta. Zanka se ponavlja toliko časa, dokler nam funkcija ne vrne praznega niza. Ker funkcija vrača vse mape, nam lahko vrne tudi simbolični poti `.` (trenutna mapa) in `..` (nadrejena mapa). Prvo

[27] Za programerja so mape pravzaprav datoteke s posebnim atributom.

vrednost zanemarimo, drugo pa bomo na seznamu prikazali zato, da bomo uporabniku omogočili hiter prehod v nadrejeno mapo. Za vsako vrnjeno datoteko moramo posebej preveriti, ali gre za mapo ali za navadno datoteko. To postori izraz:

```
GetAttr(strPot(intStran) & strDatoteka) And vbDirectory
```

Funkcija **GetAttr** vrne atribute, zakodirane v številski vrednosti. Posamezen atribut lahko izluščimo z bitnim operatorjem And in ustrezno vgrajeno konstanto. Zgornji izraz je resničen (beri: različen od nič), če ime datoteke predstavlja mapo. Vsako najdeno mapo dodamo na seznam s predpono \, druge datoteke pa neposredno. Ker imata oba seznama nastavljeno lastnost Sorted, bodo datoteke urejene po imenih, z mapami na vrhu (ker je znak \ po abecedi ASCII pred vsemi črkami).

Da bi zgled lahko preizkusili, moramo le še poklicati proceduro s pravega mesta. To je lahko odziv na dogodek Load:

```
Private Sub Form_Load()
  ' Preprosta različica

  ' Pregledamo pot
  PreglejPot "c:\", 0
  PreglejPot "d:\", 1

  ' Uskladimo seznama pogonov
  Drive1(0).Drive = strPot(0)
  Drive1(1).Drive = strPot(1)

End Sub
```

V tej, najpreprostejši obliki predpostavimo, da sta v računalniku tako pogon c: kakor d:. Kaj pa če ima uporabnik vgrajen samo en pogon?

```
Private Sub Form_Load()
  ' Malo bolj zapletena različica

  Drive1(0).Drive = "c"
  PreglejPot "c:\", 0
  If Drive1(0).ListIndex < Drive1(0).ListCount - 1 Then
    Drive1(1).Drive = "d"
    PreglejPot "d:\", 1
  Else
    Drive1(1).Drive = "c"
    PreglejPot "c:\", 1
  End

  ' Uskladimo seznama pogonov
  Drive1(0).Drive = strPot(0)
  Drive1(1).Drive = strPot(1)

End Sub
```

Pri ugotavljanju števila pogonov si lahko pomagamo kar z njihovim seznamom. Če je pogon c: že zadnji na seznamu, bomo tudi desni seznam postavili nanj. Na koncu z določeno potjo uskladimo vsebino obeh seznamov pogonov. Vemo že, da se pri nastavljanju lastnosti Drive uporabi le prvi znak niza.

NAMIG Seveda smo v obeh primerih predpostavili, da sta pogona pripravljena. Lahko bi bil pogon d: tudi CD-ROM, brez vstavljenega medija. Težavo bi lahko rešili s prestrezovalnikom napak, ki ga bomo spoznali v naslednjem poglavju.

Kodo, razvito do te točke, najdete na priloženi plošči CD-ROM, v mapi: \Primeri\2-Delavnica\1-Komandant\3-Koda.

Nadziranje nizov

Vsebina seznama odgovarja na začetku izbrani poti, vendar se zaenkrat še ne moremo sprehajati po pogonih in mapah. Prva nadgradnja programa bo uskladitev vsebine seznama pogonov z vsebino seznama map in datotek. Ustrezen dogodek že poznamo, ves podprogram pa se glasi:

```
Private Sub Drive1_Change(Index As Integer)
  PreglejPot UCase(Left(Drive1(Index).Drive, 2)), Index
End Sub
```

Kadarkoli se spremeni vsebina seznama, posredujemo proceduri PreglejPot prva dva znaka lastnosti Drive. Funkcija **Left** izlušči podano število znakov z leve strani niza, funkcija **UCase** pa jih pretvori v velike črke (to je zgolj estetski učinek). Samodejno posredovani parameter Index opredeli stran, ki jo uporabljamo. Tu se nam zelo izplača uporabiti polje gradnikov.

Sprehod po mapah bo malo bolj zapleteno opravilo. Uporabnik lahko mapo izbere na dva načina:

- Z miško dvakrat klikne ime mape na seznamu.
- Na seznamu izbrano mapo potrdi s pritiskom na tipko Enter.

Torej bomo morali uporabiti dve odzivni proceduri:

```
Private Sub List1_KeyPress(Index As Integer, KeyAscii As Integer)
  If KeyAscii = 13 Then UporabiDatoteko Index
End Sub

Private Sub List1_DblClick(Index As Integer)
  UporabiDatoteko Index
End Sub
```

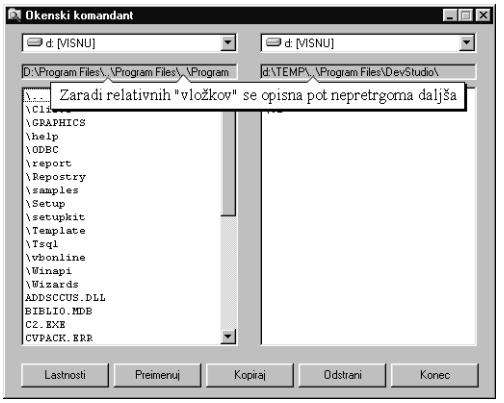
Ker v obeh primerih opravimo enako delo, je smiselno, da zapišemo nov lasten podprogram:

```
Private Sub UporabiDatoteko(intStran As Integer)
Dim strNovaPot As String
  With List1(intStran)
    If Left(.List(.ListIndex), 1) = "\" Then
      ' Če gre za mapo, se prestavi vanj
      strNovaPot = strPot(intStran) & Mid(.List(.ListIndex), 2)
      PreglejPot strNovaPot, intStran
```

```
    End If
  End With
End Sub
```

Podprogram je pravzaprav preprost. Najprej izluščimo prvi znak trenutno izbrane vrstice (s funkcijo Left). Če gre za mapo, jo uporabimo. Vendar moramo paziti. V spremenljivki `strPot(intStran)` je pot prav gotovo že končana z znakom `\` (to nam zagotavlja procedura PreglejPot, pri kateri je bila vsebina spremenljivke nazadnje popravljena), zato ga moramo odrezati iz prebrane vrstice. Tu nam pomaga funkcija **Mid**, ki vrne poljuben podniz iz posredovanega niza. Določimo ga z zaporedno številko začetnega in končnega znaka. Če končnega znaka ne podamo, se privzame zadnji znak niza. V nizih je prvi znak označen z zaporedno številko 1! V zgornjem zgledu torej uporabimo vse znake, razen prvega.

Slika 8-5: Težava pri izpisani poti



Zgled lahko preizkusimo. Morda nas bosta zmotili dve podrobnosti:

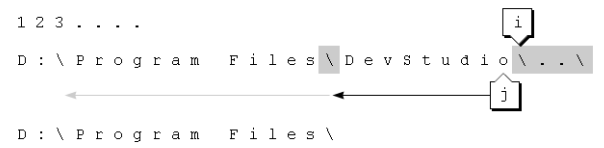
- Ponazorjena pot v oznaki se nenehno daljša.
- Če je v mapi veliko datotek, se vsebina seznama dolgo ne prikaže.

Obe težavi lahko odpravimo v proceduri PreglejPot. Njena končna oblika je razvidna spodaj, dodani deli so označeni krepko:

```
Private Sub PreglejPot(strNovaPot As String, intStran As Integer)
Dim i As Integer, j As Integer
Dim strDatoteka As String

  ' odstrani relativne premike
  If Right(strNovaPot, 1) <> "\" Then strNovaPot = strNovaPot & "\"
  i = InStr(1, strNovaPot, "\..\")
  If i > 3 Then
    For j = i - 1 To 3 Step -1
      If Mid(strNovaPot, j, 1) = "\" Then
        strNovaPot = Left(strNovaPot, j)
        Exit For
      End If
```

```
    Next
  End If

  strPot(intStran) = strNovaPot
  Label1(intStran).Caption = strPot(intStran)
  List1(intStran).Clear
  i = 0

  ' Najdemo morebitno prvo datoteko, zahtevamo tudi mape
  strDatoteka = Dir(strPot(intStran) + "*.*", vbDirectory)

  ' Ponavljamo, dokler imamo kako datoteko
  Do While strDatoteka <> ""

    ' Zanemarimo trenutno mapo, ki jo ponazarja .
    If strDatoteka <> "." Then
      ' Ali gre za mapo?
      If (GetAttr(strPot(intStran) & strDatoteka) And vbDirectory) = _
        vbDirectory Then
        ' Mape predstavimo z začetnim \
        List1(intStran).AddItem "\" & strDatoteka
      Else
        ' Preostale datoteke uvrstimo neposredno
        List1(intStran).AddItem strDatoteka
      End If
    End If

    ' Tu in tam vsilimo osvežitev gradnika,
    ' da program ni videti, kakor bi obvisel
    i = (i + 1) Mod 100
    If i = 0 Then List1(intStran).Refresh

    ' Zahtevamo naslednjo datoteko
    strDatoteka = Dir
  Loop

End Sub
```

Če bomo vstavljene relativne poti (..) pravočasno odstranjevali, bo v nizu kvečjemu ena. Mesto, na katerem se prikaže, nam najde funkcija InStr. Funkcija najde prvo pojavitev podniza v podanem nizu. Od tega mesta se v zanki For pomikamo nazaj, dokler ne najdemo nadrejene mape (odmik .. pravzaprav pomeni prejšnjo mapo, zato je zgled deloval brezhibno). Brž ko jo najdemo, popravimo shranjeno pot in nemudoma izstopimo iz zanke.



Slika 8-6: Shematska ponazoritev iskanja nadrejene mape

Druga moteča podrobnost je predolgo čakanje ob praznem seznamu, če mora procedura PreglejPot pregledati mapo z mnogo datotekami (npr. osnovno ali

sistemsko mapo samih Oken). Tu nam pomaga metoda Refresh. Če bi jo poklicali po vsaki dodani datoteki, bi vzela preveč časa, zato s pomožno spremenljivko (uporabimo kar `i`, ki nam je pomagal že pri iskanju mape) sprožimo osvežitev seznama po vsakih 100 dodanih elementih. Zanimivo, da je program zaradi tega malo počasnejši, vendar se uporabniku zdi, da se nekaj dogaja, zato je njegov subjektivni vtis boljši.

Uporaba miške in tipkovnice

Najosnovnejšo funkcionalnost program zdaj ima. Za uporabnika bo nedvomno uporabnejši, če mu dodamo še nekaj ukaznih tipk in bolje izkoristimo miško. Prej pa se posvetimo še manjši lepotni podrobnosti. Dobro bi namreč bilo, ko bi kako ponazorili izbrano stran v vmesniku. Za to bi bili najboljši oznaki, ki opisujeta izbrano pot. Ker bomo stran zamenjali ob več priložnostih, si najprej zapišimo novo lastno proceduro:

```
Private Sub ZamenjajStran(intNovaStran As Integer)
  ' Odznači prej izbrano stran
  Label1(intStran).BackColor = vbWindowBackground
  Label1(intStran).ForeColor = vbWindowText
  ' Določi novo stran
  intStran = intNovaStran
  ' Označi novo stran
  Label1(intStran).BackColor = vbHighlight
  Label1(intStran).ForeColor = vbHighlightText
End Sub
```

Procedura z barvo ozadja in črk označi izbrano stran. Aha, da bi delovalo, moramo vedeti, katera stran je bila izbrana prej. Zato potrebujemo novo modulsko spremenljivko `intStran`. Del javnih napovedi je sedaj videti takole (nova vrstica je izpisana krepko):

```
Option Explicit
Private strPot(1) As String
Private intStran As Integer
```

Kje vse bomo klicali proceduro? Prav gotovo ob začetku dela, da nastavimo spremenljivko `intStran` in uredimo barve oznak. Hkrati moramo stran zamenjati, kadarkoli prejme vnosno točko seznam pogonov ali seznam datotek in map. Kaj pa sama oznaka? Ta vnosne točke ne more prejeti, vendar prepozna dogodka Click in MouseDown. Hkrati bi bilo dobro poskrbeti, da ob izbiri strani vnosno točko nemudoma prejme sam seznam. Hm, tu pa naletimo na težavo. Vnosne točke namreč ni mogoče spreminjati, dokler obrazec ni prikazan. Torej vse spremembe iz dogodka Load ne smejo spremeniti vnosne točke. Rešitev je uporaba dogodka Activate, ki se pripeti šele po prikazanem obrazcu.

Zapišemo štiri nove odzivne procedure:

```
Private Sub Form_Activate()
  ' Uskladi barvo oznak in določi začetno stran
  intStran = 1
  ZamenjajStran 0

  ' Postavi vnosno točko na prvi seznam
  List1(0).SetFocus
End Sub

Private Sub Drive1_GotFocus(Index As Integer)
  ZamenjajStran Index
End Sub

Private Sub List1_GotFocus(Index As Integer)
  ZamenjajStran Index
End Sub

Private Sub Label1_MouseDown(Index As Integer, Button As Integer, _
Shift As Integer, X As Single, Y As Single)
  ZamenjajStran Index
  List1(Index).SetFocus
End Sub
```

V dogodku Activate najprej nastavimo desno stran za privzeto, ki jo nemudoma spremenimo v levo. Tako že na začetku uskladimo barve obeh oznak. Stran zamenjamo še v treh primerih: ko kateri izmed seznamov prejme vnosno točko in uporabnik nad oznako pritisne tipko miške. Pri seznamu pogonov dogodek Change ni primeren, saj se ne pripeti, če uporabnik izbere s seznama znova isti pogon. Pri oznaki bi bil sicer ustrezen tudi dogodek Click, vendar se MouseDown pripeti prej, kar da uporabniku dober občutek o odzivnosti programa. Kakor vidimo, se odzivni proceduri tega dogodka posreduje še nekaj parametrov (stanje gumbov, posebnih tipk in koordinata kazalca), ki pa jih zaenkrat ne bomo uporabili, na vrsto pridejo v naslednjih poglavjih tega dela knjige.

Posvetimo se sedaj izbiranju pogonov in datotek s tipkami. Dogodek KeyPress za nas ni več zadosten, saj ne prepozna pomembnih tipk, kot so npr. funkcijske. Katere tipke bi sploh želeli uporabiti v Okenskem komandantu? Sodeč po našem vzorniku so to:

Tipka	Pomen
Enter	Izbor datoteke.
Ins	O(d)značevanje datotek.
Tab	Zamenjava strani.
F2, F6	Preimenuj.
F5	Kopiraj.
F8	Briši.

Tipka	Pomen
Ctrl+F1	Izbor pogona leve strani.
Ctrl+F2	Izbor pogona desne strani.
Esc	Konec dela.

Dodali smo še tipko F2, ki je v Oknih standardna tipka za preimenovanje, ter tipko Esc, ki konča izvajanje programa.

Zelo pomembno: na obrazcu lahko tipke obdelamo na dva načina:

1. Vsaka tipka se posreduje gradniku, ki ima vnosno točko (privzeta nastavitev).
2. Vsaka tipka se najprej posreduje obrazcu, šele zatem morebitnemu gradniku z vnosno točko. V tem primeru moramo obrazcu nastaviti lastnost KeyPreview na resnično vrednost (kar smo storili tudi mi).

V našem zgledu bomo večino tipk obdelali na ravni obrazca. Samo tipko Ins bomo obravnavali le, ko je vnosna točka na seznamu. Zato bomo zapisali dve odzivni proceduri za dogodek KeyDown:

```
Private Sub List1_KeyDown(Index As Integer, KeyCode As Integer, _
Shift As Integer)
  With List1(Index)
    Select Case KeyCode
      Case 45 ' Ins
        .Selected(.ListIndex) = Not .Selected(.ListIndex)
        If .ListIndex < .ListCount - 1 Then .ListIndex = .ListIndex + 1
    End Select
  End With
End Sub

Private Sub Form_KeyDown(KeyCode As Integer, Shift As Integer)
  Select Case KeyCode
    Case 9
      ' TAB - zamenjaj stran
      ZamenjajStran (1 - intStran)
      List1(intStran).SetFocus
    Case 27
      ' Esc - zapri program
      Unload Me
    Case 112
      ' F1 - če še Ctrl, potem prvi pogon:
      If Shift And vbCtrlMask Then
        Drive1(0).SetFocus
        SendKeys "{F4}"
      End If
    Case 113
      ' F2 - če še Ctrl, potem drugi pogon, sicer preimenuj
      If Shift And vbCtrlMask Then
        Drive1(1).SetFocus
        SendKeys "{F4}"
```

```
      Else
        ' F2 - preimenuj
        ObdelajDatoteke 1
      End If
    Case 116
      ' F5 - kopiraj
      ObdelajDatoteke 2
    Case 117
      ' F6 - preimenuj
      ObdelajDatoteke 1
    Case 119
      ' F8 - briši
      ObdelajDatoteke 3
    Case Else
      ' Odstranimo komentar za analizo drugih tipk
      ' MsgBox KeyCode & "(" & Shift & ")"
  End Select
End Sub
```

Odzivna procedura dogodka KeyDown vedno prejme še dva parametra. Njun pomen je razložen v spodnji tabeli:

Parameter	Pomen
KeyCode	Koda tipke na tipkovnici. Ta koda ni nujno enaka kodi ASCII, ki jo prejme dogodek KeyPress.
Shift	Koda, ki označuje stanje nadzornih tipk: dvigalka (Shift), nadzor (Ctrl) in alternativa (Alt).

Z obema parametroma lahko zaznamo vsako tipko na tipkovnici in kombinacije s tremi nadzornimi tipkami. Nekaj zgledov:

Kombinacija	Vrednost KeyCode	Vrednost Shift
a	65	0
A	65	1
Ctrl	17	2
Ins	45	0
Enter	13	0
Shift+Alt+F1	112	5

Stanje posamezne nadzorne tipke lahko določimo z bitnim operatorjem And in vgrajeno konstanto, ki ji pripada. To so:

Tipka	Vgrajena konstanta
Shift	vbShiftMask
Ctrl	vbCtrlMask
Alt	vbAltMask

NAMIG Parameter Shift ima popolnoma enak pomen tudi pri dogodku MouseDown. Z njim ugotavljamo kombinacije posebnih tipk in pritisnjenih gumbov miške.

Zdaj, ko poznamo kode tipk, se posvetimo analizi kode. V odzivu na dogodek KeyPress znotraj seznama datotek in map smo uporabili stavek Select Case zgolj zaradi lažje nadgradnje programa (če bomo v prihodnosti želeli obravnavati še kako tipko), popolnoma bi zadoščal tudi stavek If. Ko je pritisnjena tipka Ins, spremenimo označenost izbrane vrstice in se pomaknemo navzdol, če nismo v zadnji vrstici.

Obdelava tipk za ves obrazec je obsežnejša. Tipka Tab povzroči zamenjavo strani. Novo stran izračunamo iz obstoječe s preprostim izrazom, ki 1 preslika v 0 in nasprotno. Tipko Esc lahko obdelamo tako v dogodku KeyPress kakor v KeyDown, njena koda je enaka (enako velja tudi za tipko Enter). Ponovimo še enkrat, da je stavek Unload Me edini pravilni konec dela.

Funkcijska tipka F1 je uporabna le v kombinaciji s tipko Ctrl, zato preverimo stanje parametra Shift. Če gre za kombinacijo Ctrl+F1, prestavimo vnosno točko na ustrezni seznam pogonov (zelo prav nam pride posredovani parameter Index, zaradi polja gradnikov), ki ga tudi *razpremo*. Ta učinkoviti dodatek, ki olajša nadzor Okenskega komandanta s tipkami, smo dosegli s stavkom **SendKeys**.

NAMIG S stavkom SendKeys lahko kateremukoli dejavnemu *okenskemu* programu pošljemo katerokoli kombinacijo tipk, razen tipke Print Screen. Natančen opis stavka, skupaj s kombinacijami za posebne tipke, je na voljo v sprotni pomoči. Tu razložimo le, kaj je dejavni program (*active application*). Podobno kakor je na obrazcu dejaven tisti gradnik, ki ima vnosno točko, je tudi med vsemi pognanimi programi dejaven tisti, ki mu pripada vnosna točka. Tak je vedno natančno eden. V kodi visual basica je to kar naš program, razen če prej s stavkom **AppActivate** ne predamo vnosne točke drugemu programu. V zgornjem zgledu smo tipko poslali kar seznamu pogonov, ki se je zaradi tega razprl.

Za predah si oglejmo zgled za nadziranje računala (*calculator*). Zgled predpostavlja, da je računalo nameščeno in nekje na sistemski poti.

```
Private Sub Command1_Click()
On Error GoTo Pozeni
  AppActivate "Calculator"
On Error GoTo 0
  SendKeys "{DEL}{2 3}{+}3~", True  '222 + 3
  SendKeys "^C", True               'Ctrl+C, Copy
  SendKeys "%{F4}", True            'Alt+F4, Exit
  MsgBox "222 + 3 = " & Clipboard.GetText
  Exit Sub

Pozeni:
  Shell "CALC.EXE", vbNormalFocus
  Resume Next
End Sub
```

Vsekakor je pošiljanje tipk zanimiva možnost izmenjevanja podatkov, vendar so v Oknih za medprocesno komunikacijo na voljo mnogo bolj prefinjeni protokoli, ki jih bomo spoznali v tretjem delu knjige.

Vse druge tipke smo predali naprej proceduri ObdelajDatoteke, ki jo moramo še razviti. Za začetek naj samo nakaže, kaj ji je storiti:

```
Private Sub ObdelajDatoteke(intUkaz As Integer)

  Select Case intUkaz
    Case 0 ' Lastnosti
      MsgBox "Lastnosti..."

    Case 1 ' Preimenuj
      MsgBox "Preimenuj..."

    Case 2 ' Kopiraj
      MsgBox "Kopiraj..."

    Case 3 ' Odstrani
      MsgBox "Odstrani..."

    Case 4 ' Konec
      MsgBox "Konec..."

  End Select

End Sub
```

Po preizkusu nas znova moti nekaj malenkosti. Verjetno je najhuje, da program ne uboga naših navodil za tipko Tab. Vemo, da ima ta tipka vnaprej določen pomen: prenaša vnosno točko med gradniki, ki so jo sposobni sprejeti. Če jo želimo zlorabiti v lastne namene, moramo vsem gradnikom onemogočiti lastnost TabStop. To bi sicer lahko postorili med načrtovanjem, vendar bomo raje napisali bolj splošno rešitev kar v odzivni proceduri dogodka Load. Dopolnimo jo do naslednje oblike:

```
Private Sub Form_Load()
Dim cnt As Control
```

```
  ' Izklopi tab
  For Each cnt In Controls
    If Not TypeOf cnt Is Label Then cnt.TabStop = False
  Next

' Pregledamo pot
  PreglejPot "c:\", 0
  PreglejPot "d:\", 1

  ' Uskladimo seznama pogonov
  Drive1(0).Drive = strPot(0)
  Drive1(1).Drive = strPot(1)
End Sub
```

Z zanko For Each smo se sprehodili prek vseh gradnikov. Če posamezen gradnik ni oznaka, mu onemogočimo lastnost TabStop. Oznaka te lastnosti namreč nima. S tem dodatkom tipka Tab začne ubogati naše napotke.

Do druge težave pride pri souporabi miške in tipkovnice za označevanje datotek. Brž ko z miško kliknemo seznam, se izbrana vrstica označi. Vse lepo in prav, vendar ko začnemo s tipko Ins izbirati dodatne datoteke, se že izbrana odznači. Seveda, tipka Ins pravzaprav preklaplja z izbranosti na neizbranost in nasprotno. Težavo odpravimo tako, da se ustrezno odzovemo na klik miške po seznamu:

```
Private Sub List1_MouseDown(Index As Integer, Button As Integer, _
Shift As Integer, X As Single, Y As Single)
Dim i As Integer
  If Shift = 0 Then
    With List1(Index)
      For i = 0 To .ListCount - 1
        .Selected(i) = False
      Next
    End With
  End If
End Sub
```

Klik z miško se sedaj vede podobno kakor v našem vzorniku. Enostaven klik odznači vse morebiti izbrane datoteke. Če klikamo skupaj s pritisnjeno nadzorno tipko (nismo po nepotrebnem zapletali, dobra je katerakoli: Shift, Ctrl ali Alt), se zaradi lastnosti MultiSelect na seznamu samodejno izbirajo nove vrstice.

Še ena malenkost. Tipka Enter zaenkrat odpira mape, nismo pa določili, kakšna je njena vloga pri izbrani datoteki. Recimo, da bo prikazala njene lastnosti. Uredimo tudi, da bo pritisku na tipko Enter enakovreden dvakratni klik seznama z miško. Najprej popravimo že znano lastno proceduro UporabiDatoteko:

```
Private Sub UporabiDatoteko(intStran As Integer)
Dim strNovaPot As String
  With List1(intStran)
    If Left(.List(.ListIndex), 1) = "\" Then
      ' Če gre za mapo, se prestavi vanj
      strNovaPot = strPot(intStran) & Mid(.List(.ListIndex), 2)
      PreglejPot strNovaPot, intStran
```

```
    Else
      ' Za datoteko preglej lastnosti
      ObdelajDatoteke 0
    End If
  End With
End Sub
```

Zatem dodajmo še eno odzivno proceduro:

```
Private Sub List1_DblClick(Index As Integer)
  UporabiDatoteko Index
End Sub
```

Da bo program zaključen, moramo še dokončno kodirati procedure ObdelajDatoteke.

> **NAMIG** Vso kodo, razvito do te točke, najdete na priloženi plošči CD-ROM, v mapi: \Primeri\2-Delavnica\1-Komandant\4-Koda.

Shranjevanje nastavitev

Preden se spopademo z datotekami, si oglejmo še eno izboljšavo Okenskega komandanta. Lepo bi bilo, ko bi nas program ob vnovičnem zagonu pričakal tam, kjer smo zadnjič končali. Sedaj v odzivni proceduri dogodka Load vedno nastavimo pot na izhodiščno mapo pogona c: oziroma d:.

Za to dodatno zmožnost moramo uporabiti vgrajena podprograma **GetSetting** in **SaveSetting**. Posodobili bomo odzivno proceduro dogodka Load obrazca in zapisali še odziv na dogodek Unload:

```
Private Sub Form_Load()
Dim cnt As Control

  ' Izklopi tab
  For Each cnt In Controls
    If Not TypeOf cnt Is Label Then cnt.TabStop = False
  Next

  ' Preberi nastavitve
  strPot(0) = GetSetting("Komandant", "Nastavitve", "PotLevo", "C:\")
  strPot(1) = GetSetting("Komandant", "Nastavitve", "PotDesno", "C:\")

  Drive1(0).Drive = strPot(0)
  Drive1(1).Drive = strPot(1)

  PreglejPot strPot(0), 0
  PreglejPot strPot(1), 1

End Sub

Private Sub Form_Unload(Cancel As Integer)
  SaveSetting "Komandant", "Nastavitve", "PotLevo", strPot(0)
  SaveSetting "Komandant", "Nastavitve", "PotDesno", strPot(1)
End Sub
```

Funkcija GetSetting prebere nastavitev, ki je shranjena v *registru*. Register (*registry*) je skupina vseh sistemskih nastavitev[28]. Kot parametre ji posredujemo štiri nize: naziv programa, naziv drevesa nastavitev, naziv nastavitve in privzeto vrednost te nastavitve, če je v registru ni mogoče prebrati. Procedura SaveSetting shrani želeno nastavitev. Prvi trije parametri imajo enak pomen, zadnji pa je dejanska vrednost nastavitve. Za delo z registrom imamo na voljo še funkcijo **GetAllSettings**, ki vrne vse nastavitve v izbranem drevesu v obliki dvodimenzionalnega polja znotraj tipa Variant, in funkcijo **DeleteSetting**, ki lahko zbriše posamezno nastavitev ali celo drevo.

> **NAMIG** Izhodišče v registru, pod katerim se shranjujejo naše nastavitve, je ključ \HKEY_CURRENT_USER\Software\VB and VBA Program Settings\.

Vendar brez težav tudi tokrat ne gre. Zalomi se pri obnovitvi poti. Videti je, da je Okenski komandant uspel obnoviti pogon, nekako pa mu ne uspe, da bi se postavil še v pravo mapo. Le v čem je težava?

Iskanje napake

Orodje, brez katerega je programiranje praktično nemogoče, je prav gotovo razhroščevalnik. Visual basic ima vgrajenega enega najboljših sploh. Ker se koda v izvajalnem okolju vedno tolmači, je mogoče program med izvajanjem neboleče zaustaviti, opazovati vrednosti spremenljivk, kako tudi popraviti, in ga vnovič pognati.

Pred iskanjem hroščev najprej obnovimo težaven zgled. Poženemo Okenski komandant in se na levem in desnem seznamu prebijemo kam globoko v datotečno strukturo. Zatem končamo delo z izbiro gumba »Konec«.

Da bi si delo olajšali, najprej odprimo dodatno vrstico z gumbi, namenjeno prav izločanju napak iz kode. Prikažemo jo z ukazom **View ➪ Toolbars ➪ Debug**, lahko pa jo vklopimo tudi s priročnim menujem osnovne vrstice z gumbi.

Slika 8-7: Vrstica z gumbi za lažje razhroščevanje

Program najpogosteje razhroščujemo tako, da pred sumljivo mesto v kodi postavimo prekinitveno točko (*breakpoint*) in zatem nadaljujemo izvajanje korak za korakom ob

[28] V Oknih 3.1 so bile nastavitve večinoma shranjene v datotekah INI, register pa so uporabljali le predmeti OLE. V Oknih 95 je register osrednja shramba vseh nastavitev. Nahajajo se v datotekah SYSTEM.DAT in USER.DAT, ki sta posebej zaščiteni in ju vzdržujejo izključno Okna.

spremljanju vrednosti določenih spremenljivk. Prekinitvena točka je lahko postavljena pred katerikoli programski stavek, lahko pa jo nastavimo tudi pogojno. Pogojna prekinitvena točka se sproži, kadar se med izvajanjem spremeni vrednost izbrane spremenljivke ali izraza.

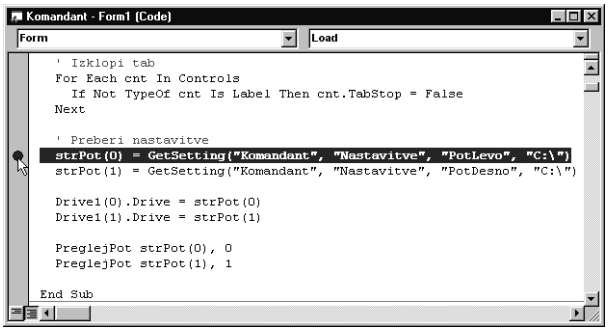
Prekinitveno točko bomo postavili v vrstico znotraj prve procedure, ki se izvede. Postavitev prekinitvene točke je mogoča na več načinov. Najprej se postavimo v programski stavek, pred katerim bi radi prekinili izvajanje. Zatem izberemo eno izmed možnosti:

- Pritisnemo tipko F9.
- Izberemo ukaz **Set Breakpoint** z glavnega menuja **Debug**.
- Izberemo **Toggle ➪ Breakpoint** s priročnega menuja urejevalnika.
- Kliknemo gumb **Toggle Breakpoint** v vrstici z gumbi Debug.

Vsi navedeni postopki izmenoma vklapljajo in izklapljajo (*toggle*) prekinitveno točko, ki je ponazorjena z rdečim krožcem ob strani, rdeče pa je obarvana tudi vrstica kode, v kateri je. Kar nas pripelje do še enega in verjetno najlažjega načina preklapljanja prekinitvene točke:

- Z miško kliknemo sivi stolpec levo od stavka, v katerem želimo prekinitveno točko.

Slika 8-8: Postavljanje prekinitvene točke z miško



Postavimo prekinitveno točko na mesto, razvidno na zgornji sliki, torej na stavek:

```
strPot(0) = GetSetting("Komandant", "Nastavitve", "PotLevo", "C:\")
```

Prekinitveno točko lahko postavimo tudi programsko. Izvajanje programa začasno zaustavi ukaz **Stop**, ki je samostojen programski stavek v visual basicu.

Ko poženemo program, bo ta nemudoma prekinjen, izvajalno okolje pa se prestavi v razhroščevanje, kar ponazarja oznaka **[break]** v naslovni vrstici okna visual basica. Nekateri prej onemogočeni gumbi v vrstici Debug so oživeli, na voljo pa so tudi istoimenski ukazi glavnega menuja Debug. Za začetek bomo korak za korakom

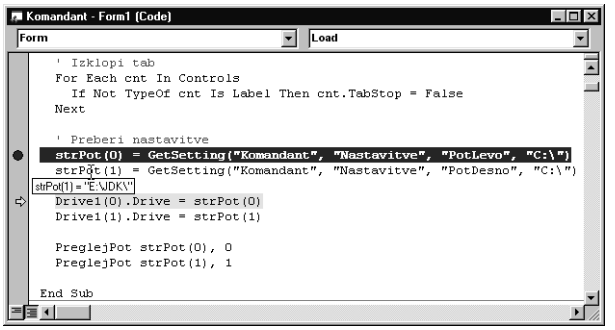
izvedli prva dva stavka, ki prebereta nastavitve. Izvajanje naslednjega stavka omogočimo na več načinov:

- Pritisnemo tipko F8.
- Izberemo ukaz **Step Into** z glavnega menuja **Debug**.
- Na voljo je tudi enakovreden gumb v vrstici Debug.

> **NAMIG** Na voljo je več načinov koračnega izvajanja. Ukaz **Step Into** je najtemeljitejši, saj se spusti tudi znotraj vseh klicanih podprogramov. Če nas ti ne zanimajo, lahko izberemo ukaz **Step Over**. Če smo z ukazom Step Into že zakorakali v podprogram, pa nas ta ne zanima več, lahko hitro pridemo nazaj, v točko po njegovem klicu, če izberemo ukaz **Step Out**.

Zdaj bi si bilo dobro ogledati vrednosti v polju `strPot`. Opravilo je tako preprosto, da je z njim razhroščevanje v visual basicu postalo celo zabavno. Enostavno z miško pokažemo izpisano ime spremenljivke, malo počakamo in v okencu z namigom bomo lahko prebrali njeno vrednost.

Slika 8-9: Opazovanje trenutne vrednosti spremenljivke v okencu z namigom



Ugotovimo lahko, da je prebrana vrednost pravilna. Nadaljujmo koračno izvajanje še za en stavek. Hej, kaj pa je to? Nenadoma smo se znašli znotraj procedure Drive1_Change! Ahhh, v tem je torej težava. Pozabili smo, da sprememba lastnosti Drive samodejno povzroči dogodek Change. Odzivna procedura tega dogodka pa pokvari izbrano pot, saj jo uskladi z novo izbranim pogonom.

Preverimo do konca. Izberimo ukaz **Debug ➪ Step Out**, tako da se ne bomo spuščali v podrobnosti podprograma PreglejPot. Novo preverjanje vsebine spremenljivke `strPot(0)` potrdi naše slutnje. Zdaj lahko program brez prekinitev poženemo naprej (**Run ➪ Continue**) ali pa izvajanje dokončno zaustavimo (**Run ➪ End**).

> **NAMIG** Z izbiro ukaza **Run ➪ End**, ki je – mimogrede – enakovreden programskemu stavku **End**, izvajanja programa ne končamo pravilno,

saj se izvajanje zaustavi nemudoma in se morebiti zasežena pomnilniška sredstva predmetov ne sprostijo. Pri programih, ki ne uporabljajo lastnih predmetov, ni posebnih težav, lahko pa do njih pride pri bolj zapletenih zgledih.

Rešitev težave je v tem, da usklajevanje seznama datotek in map s seznamom pogonov začasno preprečimo. Za to bomo potrebovali novo modulsko spremenljivko logičnega tipa:

```
Option Explicit
Private strPot(1) As String
Private intStran As Integer
Private blnUpostevajDriveList As Boolean
```

Pred usklajevanjem s seznamom pogonov znotraj procedure Form_Load to spremenljivko nastavimo:

```
Private Sub Form_Load()
...
  ' Prepreči kvarjenje poti
  blnUpostevajDriveList = False
  Drive1(0).Drive = strPot(0)
  Drive1(1).Drive = strPot(1)
...
End Sub
```

To spremenljivko mora upoštevati nekoliko spremenjena odzivna procedura dogodka Change za oba seznama pogonov:

```
Private Sub Drive1_Change(Index As Integer)
  If blnUpostevajDriveList Then
    PreglejPot UCase(Left(Drive1(Index).Drive, 2)), Index
  End If
End Sub
```

Na koncu pa moramo poskrbeti, da se bo v prihodnje sprememba pogona odrazila na seznamu datotek. To postorimo s stavkom, ki ga uvrstimo v proceduro Form_Activate:

```
Private Sub Form_Activate()
...
  ' Vklopi upoštevanje seznama pogonov
  blnUpostevajDriveList = True
...
End Sub
```

S tem smo se težavi dovolj elegantno izognili.

Drugo, še primernejše iskanje napake bi bila uporaba opazovanega izraza (*watch*). To in druga orodja za iskanje napak bomo uporabljali pri zgledih v nadaljevanju.

Upravljanje datotek

Visual basic pozna kar precej podprogramov in programskih stavkov za upravljanje datotečnega sistema (*file system administration*). Nekatere smo spoznali že med dosedanjim razvojem Okenskega komandanta, vse skupaj pa bomo pregledno našteli v naslednji tabeli:

Podprogram/ Stavek	Zgled rabe	Opis
Dir	`MsgBox Dir("C:\WINDOWS\*.*", vbDirectory)`	Najde prvo ali vse datoteke, ki ustrezajo podanemu vzorcu imena in atributov.
CurDir	`MsgBox CurDir`	Vrne dejavno pot.
ChDir	`ChDir "D:\TEMP\"`	Zamenja pot v poljubnem pogonu.
ChDrive	`ChDir "D"`	Spremeni dejavni pogon in dokončno določi dejavno pot.
RmDir	`RmDir "D:\TEMP\"`	Odstrani obstoječo *prazno* mapo v poljubnem pogonu.
MkDir	`MkDir "D:\TEMP\"`	Ustvari novo mapo znotraj *obstoječe* mape v poljubnem pogonu.
GetAttr	`MsgBox GetAttr ("C:\BOOTLOG.TXT")`	Vrne kodo atributov za podano datoteko/mapo.
SetAttr	`SetAttr "D:\TEMP\"`	Nastavi atribute za podano datoteko/mapo.
Name...As	`Name "D:\ZAPIS.TXT" As "D:\TEMP\ZAPIS.BAK"`	Preimenuje in/ali premesti obstoječo datoteko v drugo, obstoječo mapo v *istem* pogonu.
Kill	`Kill "D:\TEMP\*.BAK"`	Dokončno odstrani datoteko(e). Ne zna uporabiti koša (*recycle bin*)!
FileCopy	`FileCopy "D:\TEMP\ZAPIS.BAK" "C:\ZAPIS.TXT"`	Prepiše datoteko. Kopija je lahko v drugi mapi v drugem pogonu.
FileLen	`MsgBox FileLen("C:\WINDOWS\WIN.INI")`	Vrne dolžino datoteke v zlogih.
FileDateTime	`MsgBox FileDateTime ("C:\WINDOWS\")`	Vrne datum in uro zadnje spremembe datoteke.

Vidimo, da imamo za precej sorodna opravila enkrat na voljo podprogram, drugič pa kar programski stavek. Ta pisana druščina je predvsem posledica združljivosti

s starejšo kodo v basicu. Najbolj razveseljivo je, da imamo prav za vsa opravila, ki jih potrebujemo za Okenski komandant, že pripravljeno pomoč.

Oglejmo si programsko kodo procedure ObdelajDatoteke:

```
Private Sub ObdelajDatoteke(intUkaz As Integer)
Dim blnOznaceno As Boolean, blnDir As Boolean, blnOsvezi As Boolean
Dim strDatoteka As String, strDatoteka2 As String
Dim i As Integer

  ' Morda konec?
  If intUkaz = 4 Then
    Unload Me
    Exit Sub
  End If

  With List1(intStran)
    If .ListIndex = -1 Or _
    (.SelCount = 0 And .List(.ListIndex) = "\..") Then
      MsgBox "Izberi datoteko!", vbExclamation
      Exit Sub
    End If

    ' Če ni izbrano nič, označim vrstico z vnosno točko
    ' in si to zapomnim za poznejše odznačenje
    If .SelCount = 0 Then
      .Selected(.ListIndex) = True
      blnOznaceno = True
    Else
      blnOznaceno = False
    End If

    Select Case intUkaz
      Case 0 ' Lastnosti,
        ' Vedno uporabimo samo zadnje izbrano datoteko
        strDatoteka = .List(.ListIndex)
        MsgBox "Lastnosti za " & strDatoteka

      Case 1 To 3 ' Preimenuj, kopiraj, odstrani
        ' Uporabimo vse označene datoteke
        For i = 0 To .ListCount - 1
          If .Selected(i) Then
            strDatoteka = .List(i)
            blnDir = (Left(strDatoteka, 1) = "\")
            If blnDir Then
              strDatoteka = strPot(intStran) & Mid(strDatoteka, 2)
            Else
              strDatoteka = strPot(intStran) & strDatoteka
            End If
            Select Case intUkaz
              Case 1
                strDatoteka2 = InputBox("Novo ime za " & strDatoteka & _
                " ?", "Preimenovanje", strPot(1 - intStran))
              Case 2
                strDatoteka2 = InputBox("Kopiram " & strDatoteka & _
                " v: ", "Prepisovanje", strPot(1 - intStran))
```

```
            Case 3
              If MsgBox("Resnično odstranim " & strDatoteka & " ?", _
              vbQuestion + vbOKCancel + vbDefaultButton2, _
              "Odstranjevanje") = vbOK Then
                strDatoteka2 = "ok"
              Else
                strDatoteka2 = ""
              End If
          End Select

          ' Uspešno vneseno ime / potrjena operacija?
          If strDatoteka2 <> "" And strDatoteka2 <> strDatoteka Then
            ' Opravi nalogo in označi potrebo po osvežitvi seznamov
            Select Case intUkaz
              Case 1
                Preimenuj strDatoteka, strDatoteka2
              Case 2
                Prepisi strDatoteka, strDatoteka2
              Case 3
                Odstrani strDatoteka
            End Select
            blnOsvezi = True
          End If
        End If
      Next

  End Select

  ' Odznačim, kar sem morda označil
  If blnOznaceno Then .Selected(.ListIndex) = False

  ' Osvežim seznama, če je treba
  If blnOsvezi Then
    PreglejPot strPot(0), 0
    PreglejPot strPot(1), 1
  End If

  ' Vrnem vnosno točko, ki se je preselila na gumb, nazaj na seznam
  .SetFocus

 End With

End Sub
```

V podprogramu najprej preverimo, ali ni uporabnik morda izbral konec dela. V tem primeru končamo izvajanje programa na edini pravilni način: z odstranitvijo obrazca. Ta ukaz za razliko od nepravilnega konca s stavkom End pusti programu, da se pravilno izteče. Zato ne smemo pustiti podprogramu, da se izvaja naprej, ampak ga predčasno končamo s stavkom Exit Sub.

Naslednje preverjanje se ukvarja z vprašanjem, ali je sploh izbrana kaka datoteka in da ta morda ni oznaka za prejšnjo mapo. Na tovrstni nerodnosti opozorimo s sporočilom in znova predčasno končamo podprogram.

Če uporabnik ni posebej označeval datotek, bi verjetno rad uporabil natančno tisto, na kateri stoji. Zaradi enotnega obravnavanja v nadaljevanju to vrstico začasno označimo in si to zapomnimo, da bomo na koncu lahko stvari vrnili v prvotno stanje.

Srce podprograma je stavek Select Case. Čeprav se v osnovi veji le na dva dela – torej bi ga lahko preprosto nadomestili s stavkom If – je ta način bolj prilagodljiv za morebitne nove zmožnosti. Ker prikaz lastnosti za več datotek ni preveč smiseln, vedno uporabimo le zadnjo izbrano. Izčrpnejšo obdelavo datoteke bomo odložili za nadaljevanje, zaenkrat naj zadošča le sporočilo, ki ga izpiše podprogram MsgBox.

Drugi ukazi uporabljajo vse označene datoteke oziroma mape. Z zanko For za vsako vrstico seznama preverjamo, ali je izbrana. Če je, pravilno sestavimo ime datoteke in trenutno pot in vprašamo uporabnika za potrditev operacije. Za potrditev rabi vgrajena funkcija InputBox, ki omogoča popravek ciljne poti, le za potrditev brisanja pride v poštev dobri stari MsgBox. Obema funkcijama se bomo izčrpno posvetili v nadaljevanju. Potrjeno operacijo zgolj preložimo na ramena novih lastnih procedur: Preimenuj, Prepisi in Odstrani. Po koncu dela je treba odznačiti morebitno dodatno označitev, osvežiti vsebino seznamov in ponovno nastaviti vnosno točko. Zakaj? Če smo ukaz sprožili s klikom gumba, se je tja preselila tudi vnosna točka.

Hm, bi pustili izpeljavo novih lastnih procedur za vajo? Vsekakor vam priporočam, da se poizkusite v programiranju. Če boste obupali ali celo nimate dovolj poguma, da bi se lotili dela, pa berite naprej.

Napisali bomo nekaj uporabnih podprogramov, zato ni preveč smiselno, da jih zapiramo v zasebni del obrazca. Bolje bo, da zanje odpremo posebni modul s kodo. S tem pridobimo dvoje:

- Podprogrami bodo na voljo globalno.
- Modul s kodo lahko brez težav vključimo v nove projekte.

Nov modul dodamo z ukazom **Add Module** z glavnega menuja **Project**. Prikaže se galerija predlog, pri katerih potrdimo običajni, prazni modul z izbiro gumba **Open**.

Najprej bomo v modul zapisali dve splošno uporabni funkciji:

```
' Izlušči samo pot iz popolne poti do datoteke:
' c:\mapa1\mapa2\mapa3\datoteka.dat
Public Function IzlusciPot(ByVal strNiz As String) As String
Dim i, j As Integer
  i = 0
  Do
    j = i
    i = InStr(i + 1, strNiz, "\")
  Loop While i > 0
  IzlusciPot = Left(strNiz, j)
End Function
```

```
' Izlušči samo ime datoteke iz popolne poti:
' c:\mapa1\mapa2\mapa3\datoteka.dat
Public Function IzlusciDatoteko(ByVal strNiz As String) As String
Dim i, j As Integer
  i = 0
  Do
    j = i
    i = InStr(i + 1, strNiz, "\")
  Loop While i > 0
  IzlusciDatoteko = Mid(strNiz, j + 1)
End Function
```

Prva funkcija iz niza, ki predstavlja popolno pot do datoteke, izlušči samo pot, druga pa samo datoteko. Če vam delovanje ni razumljivo, vam priporočam, da se s papirjem in svinčnikom sprehodite od stavka do stavka in si v tabelo zapisujete vrednosti spremenljivk. Lahko pa si pomagate tudi z razhroščevalnikom.

Preden se lotimo drugih podprogramov, pa še pomembno opozorilo! Podprogrami, ki jih bomo zapisali, še posebej podprogram, ki odstrani poljubno pot, so **zelo nevarni**! Če boste kodo prepisali ali pa jo uporabili neposredno s priložene plošče CD-ROM, se zavedajte, da jo uporabljate na **lastno odgovornost**! Niti avtor niti založba ne prevzemata nikakršne odgovornosti za morebitno škodo! Toliko v opozorilo, sedaj pa na delo...

```
Public Sub Prepisi(strF1 As String, strF2 As String)
Dim strDatoteka As String
Dim i As Integer, j As Integer
  If (GetAttr(strF1) And vbDirectory) Then
    Debug.Print "Ustvarjam mapo: " & strF2 & IzlusciDatoteko(strF1)
    ' dejavni stavek: MkDir (strF2 & IzlusciDatoteko(strF1))
    strDatoteka = Dir(strF1 & "\*.*", vbDirectory)
    i = 0
    Do Until strDatoteka = ""
      i = i + 1
      If strDatoteka <> "." And strDatoteka <> ".." Then
        Prepisi (strF1 & "\" & strDatoteka), (strF2 & IzlusciDatoteko(strF1) & "\")
        j = 0
        strDatoteka = Dir(strF1 & "\*.*", vbDirectory)
        Do
          strDatoteka = Dir
          j = j + 1
        Loop Until j = i
      Else
        strDatoteka = Dir
      End If
    Loop
  Else
    Debug.Print "Prepisujem: " & strF1 & "->" + strF2 & IzlusciDatoteko(strF1)
    ' dejavni stavek: FileCopy strF1, strF2 & IzlusciDatoteko(strF1)
  End If
End Sub
```

Pri delu z mapami, v katerih so lahko nove mape itn., je najbolj učinkovita programerska tehnika *rekurzija*. V proceduri se sprehodimo od datoteke do datoteke, navadne neposredno prepišemo, mape pa najprej ustvarimo in znova pokličemo isto proceduro s spremenjenimi parametri.

Procedura za prepis datoteke najprej preveri, ali ne gre morda za prepis mape. V tem primeru v ciljni mapi ustvari enako poimenovano mapo in sproži pregled mape z nam že znano funkcijo Dir. Vsako najdeno datoteko v mapi (če ne gre za simbolični mapi `.` in `..`) enostavno posreduje v rekurzivnem klicu s spremenjenimi parametri nazaj sama sebi. Če datoteka ni mapa, jo zgolj prepiše.

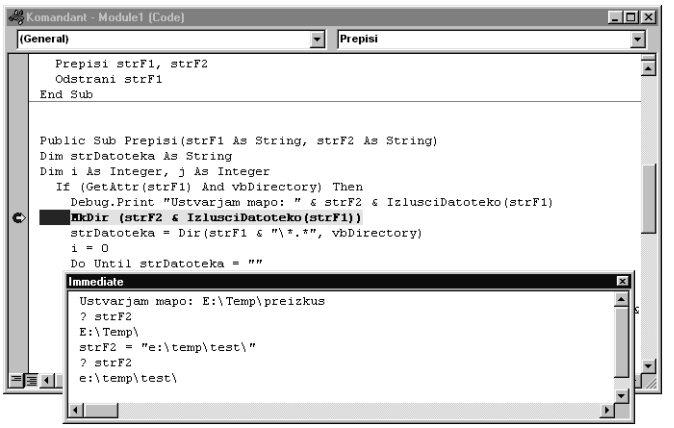
Procedura bi bila lahko še krajša, če ne bi naleteli na težave pri uporabi funkcije Dir. Kakor vemo, te funkcije prvič ne smemo poklicati brez parametrov. Zato moramo po vrnitvi iz rekurzivnega klica znova sprožiti preiskovanje. Ampak to preiskovanje se vedno začne pri prvi datoteki, ki smo jo že obdelali! Zato vpeljemo dva števca. Prvi (`i`), prešteje, koliko datotek smo že obdelali pred rekurzivnim klicem, drugi (`j`) pa poskrbi, da po vrnitvi iz rekurzivnega klica natančno toliko datotek pregledamo v prazno. S tem si zagotovimo, da se bo rekurzija na koncu srečno iztekla.

Znotraj preocedure smo uporabili še en pripomoček za razhroščevanje, vgrajeni predmet **Debug**. Ta je zelo preprost, saj pozna le dve metodi:

Metoda	Opis
`Debug.Assert logičniIzraz`	Zaustavi izvajanje, če je logični izraz resničen.
`Debug.Print niz`	Izpiše niz v okno Immediate.

Najzanimivejša lastnost predmeta Debug je, da ga najdemo zgolj v razvojnem okolju. Če program prevedemo v samostojno obliko, se vsi klici metod predmeta zanemarijo. Tako imamo lahko v programski kodi cel kup preverjanj, ki se v ničemer ne poznajo v končni različici programa. Okno Immediate, ki ga uporablja metoda Print, je prikazano ob izvajanju programa. Ročno ga lahko prikličemo z ukazom **Immediate Window** iz glavnega menuja **View**. V okno Immediate lahko vpisujemo programske stavke, ki se bodo izvedli, brž ko pritisnemo tipko Enter (od tod njegovo ime). Uporabno je za preverjanje in popravljanje vrednosti spremenljivk med razhroščevanjem, ko je izvajanje programa začasno zaustavljeno. Vrednost lahko hitro izpišemo z ukazom ? (okrajšava za stavek **Print**), popravimo pa jo z navadnim prireditvenim stavkom. Na sliki 8-10 je zgled rabe okna Immediate na zadnji proceduri.

Naslednja procedura bo poskrbela za odstranitev datoteke oziroma mape in vse njene vsebine. Je *zelo zelo* nevarna, saj se ne ustavi pred nobeno oviro (razen pred zaščiteno ali skrito datoteko). Za brisanje uporablja stavek Kill, ki ne pozna koša (*recycle bin*), zato briše brez možnosti poznejše razveljavitve.



Slika 8-10: Zgled rabe okna Immediate

```
Public Sub Odstrani(strF As String)
Dim strDatoteka As String
  If (GetAttr(strF) And vbDirectory) Then
    strF = strF & "\"
    strDatoteka = Dir(strF & "*.*", vbDirectory)
    Do Until strDatoteka = ""
      If strDatoteka <> "." And strDatoteka <> ".." Then
        Odstrani (strF & strDatoteka)
        strDatoteka = Dir(strF, vbDirectory)
      Else
        strDatoteka = Dir
      End If
    Loop
    Debug.Print "Brišem mapo: " & strF
    RmDir strF
  Else
    Debug.Print "Brišem datoteko: " & strF
    Kill strF
  End If
End Sub
```

Procedura je precej podobna prepisovanju, vendar je nekoliko preprostejša. Ker fizično odstranjuje datoteke, ni nevarnosti, da bi vnovič sproženi pregled s funkcijo Dir vračal že obdelane datoteke. To kodo uporabljajte skrajno previdno!

Kako zapletena mora biti šele procedura za preimenovanje! Stavek Name...As ne obeta nič dobrega. Hm, poizkusimo:

```
Public Sub Preimenuj(strF1 As String, strF2 As String)
  Prepisi strF1, strF2
  Odstrani strF1
End Sub
```

Zakaj bi ne uporabili, kar že imamo? Pa še varneje je tako. Ker datoteke najprej prepišemo, ni nevarnosti, da bi zaradi višje sile (denimo izpada elektrike) med premikanjem katero izgubili.

Uporaba več obrazcev

Le malo projektov lahko končamo z enim samim obrazcem. Pogosto je treba uporabnika kaj vprašati, zahtevati potrditev in podobno. Kakor smo videli že v dosedanjih zgledih, nam dostikrat ni treba pripravljati posebnega obrazca, ampak posežemo kar po dveh vgrajenih funkcijah. To sta:

Funkcija	Namen
MsgBox	Izpis sporočil, potrjevanje ukazov (funkcija upošteva jezikovno različico operacijskega sistema pri prikazu gumbov).
InputBox	Vnos enostavnih vrednosti.

Popolna oblika funkcije MsgBox je:

```
MsgBox(poziv[, vrsta][, naslov][, datotekaPomoči, oznakaPomoči])
```

Poziv je niz, ki se izpiše v oknu. Če želimo, lahko v niz vstavimo prehode v novo vrstico, ki jih ponazarja vgrajena konstanta vbCrLf. Vrsta je številčna koda, ki določa morebitno ikono in gumbe znotraj okna. Na voljo imamo cel kup konstant, ki določajo različne izpeljanke okna. Po želji jih lahko kombiniramo s seštevanjem ali bitnim operatorjem Or. Najbolj zanimive konstante so:

Konstanta	Namen
vbOKCancel	Prikazana bosta gumba OK in Cancel*.
vbYesNo	Prikazana bosta gumba Yes in No*.
vbAbortRetryIgnore	Prikazani bodo gumbi Abort, Retry in Ignore*.
vbCritical	Ponazori sporočilo z ikono napake.
vbQuestion	Ponazori sporočilo z ikono vprašanja.
vbExclamation	Ponazori sporočilo z ikono nevarnosti.
vbInformation	Ponazori sporočilo z ikono informacije.
vbDefaultButton2	Določi, da je privzet drugi gumb (neposredno ga izbere tipka Enter).

* Napisi na gumbih so ustrezni jezikovni različici oken, torej bo npr. konstanta vbYesNo pri slovenskih Oknih prikazala gumba z napisoma Da in Ne. Če parametra vrste ne podamo, se prikaže okno brez ikone, zgolj z gumbom OK.

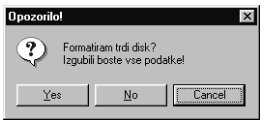
Naslednji neobvezni parameter je naslov. Gre za niz, ki se izpiše v naslovni vrstici prikazanega okna. Če ga ne podamo, se uporabi ime projekta. To ni slab izbor, saj se lahko okno z opozorilom prikaže tudi, ko je naš program v ozadju. Dobro je, da že naslovna vrstica okna pove, kateri program je povzročil njegov prikaz.

Z zadnjima dvema parametroma se lahko navežemo na datoteko s pomočjo. Več o pripravi sprotne pomoči bomo izvedeli v našem tretjem zgledu.

Podprogram MsgBox lahko uporabimo zgolj za izpis sporočila (kot proceduro), še uporabnejši pa je kot funkcija. Vrednost, ki jo vrača, lahko preverimo z vgrajenimi konstantami, ki so razumljive same po sebi, npr. vbCancel ustreza pritisnjenemu gumbu za preklic.

Zgled rabe:

```
If MsgBox("Formatiram trdi disk?" & vbCrLf & _
    "Izgubili boste vse podatke!", _
    vbQuestion + vbYesNoCancel + vbDefaultButton3, _
    "Opozorilo!") = vbYes Then
  MsgBox "Formatiram disk!", vbInformation
End If
```



Slika 8-11: Prikazano vprašanje

Druga nadvse uporabna funkcija je InputBox. Prikaže pogovorno okno z vnosnim poljem za vnašanje enostavne vrednosti, ki jo vedno vrne v obliki niza. Njena popolna oblika je:

```
InputBox(poziv[, naslov][, privzeto][, x][, y][, datotekaPomoči, oznakaPomoči])
```

Prvi parameter je niz, ki se izpiše v oknu. Podobno kakor poziv pri podprogramu MsgBox ga lahko izpišemo v več vrsticah. Vsi drugi parametri so neobvezni. Naslov predstavlja niz v naslovni vrstici okna, naslednji parameter je privzeta vsebina vnosnega polja, x in y sta zaslonski koordinati, na katerih se okno sprva prikaže, zadnja dva parametra pa smo že omenili pri podprogramu MsgBox.

Zgled rabe:

```
MsgBox InputBox("Vnesi ime!", "Vnos podatka", "(neimenovan)", 100, 100)
```

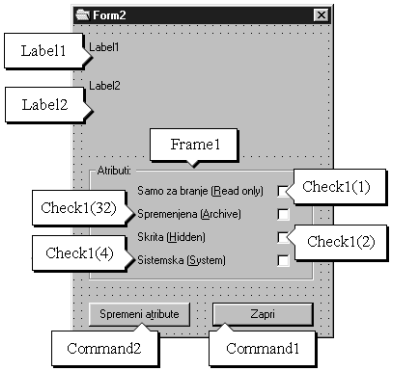
Slika 8-12: Prikazan enostavni vnos

Funkcija vrne prazen niz, če je uporabnik preklical vnos s pritiskom na tipko Cancel. Tako ne moremo ugotoviti, ali je dejansko želel vnesti prazen niz ali preklicati operacijo. To ni prevelika težava, saj s praznim nizom pogosto ne moremo početi nič pametnega.

Obe funkciji smo precej izčrpno uporabljali pri razvoju Okenskega komandanta. Za konec dela pa potrebujemo nekaj zahtevnejšega: okno za prikaz in spreminjanje lastnosti datoteke. Zato se bomo morali lotiti načrtovanja in kodiranja novega obrazca.

Nov obrazec dodamo v projekt z izbiro ukaza **Add Form** z glavnega menuja **Project**. Prikaže se galerija predlog, pri katerih je že izbran najosnovnejši obrazec. Potrdimo ga z gumbom Open. Novi obrazec je v našem projektu dobil privzeto ime Form2, ki ga ne bomo spreminjali. Delo začnemo z risanjem vmesnika, ki ga pripravimo po zgledu s spodnje slike:

Slika 8-13: Vmesnik pomožnega okna za prikaz lastnosti datoteke



Spodnja tabela prikazuje nastavitve lastnosti posameznih gradnikov:

Gradnik	Lastnost	Nastavitev
obrazec (*Form*)	(Name)	Form2
	BorderStyle	3 - Fixed Dialog
	Icon	(datoteka Form2.ico)[29]
	StartUpPosition	2 - CenterScreen
oznaka (*Label*)	(Name)	Label1
oznaka (*Label*)	(Name)	Label2
okvir (*Frame*)	(Name)	Frame1
	Caption	"Atributi: "
stikalo (*CheckBox*)	(Name)	Check1
	Caption	"Samo za branje (&Read only)"

[29] Datoteke binarnih sredstev lahko najdete na priloženi plošči, v mapi \Primeri\2-Delavnica\1-Komandant\5-Koda. Ikone so povzete iz visual basicu priložene zbirke grafičnih elementov, le da so preimenovane v ime gradnika, ki mu pripadajo. Ikona v ničemer ne vpliva na preostanek zgleda, zato lahko izberete tudi katero koli drugo.

Gradnik	Lastnost	Nastavitev
	Index	1
	Alignment	1 – Right Justify
stikalo (*CheckBox*)	(Name)	Check1
	Caption	"Spremenjena (&Archive)"
	Index	32
	Alignment	1 – Right Justify
stikalo (*CheckBox*)	(Name)	Check1
	Caption	"Skrita (&Hidden)"
	Index	2
	Enabled	False
	Alignment	1 – Right Justify
stikalo (*CheckBox*)	(Name)	Check1
	Caption	"Sistemska (&System)"
	Index	4
	Enabled	False
	Alignment	1 – Right Justify
gumb (*CommandButton*)	(Name)	Command1
	Caption	"Zapri"
	Cancel	True
	Default	True
gumb (*CommandButton*)	(Name)	Command2
	Caption	"Spremeni a&tribute"

Posebno omembo si zaslužijo zaporedna števila stikal (lastnost Index), ki jih uvrstimo v polje gradnikov. Omenili smo že, da nam lahko rabijo kot nosilci dodatne informacije. V našem zgledu smo jim priredili enako vrednost, kakor pripada ustrezni vgrajeni konstanti datotečnega atributa. To nam bo precej olajšalo poznejše kodiranje. Omenimo še ukazni gumb Command1, ki ima lastnosti Cancel in Default nastavljeni na resnično vrednost. To pomeni, da ga izbere pritisk bodisi na tipko Esc bodisi na tipko Enter.

Lotimo se še programske kode. Najprej bomo morali dokončati glavno križišče na osnovnem obrazcu, proceduro ObdelajDatoteko. Ta je doslej za lastnosti datoteke prikazala zgolj sporočilo z imenom datoteke. Popravljeni segment podprograma je sedaj videti takole:

```
Private Sub ObdelajDatoteke(intUkaz As Integer)
...
    Select Case intUkaz
      Case 0 ' Lastnosti,
```

```
        ' Vedno uporabimo samo zadnjo izbrano datoteko
        strDatoteka = .List(.ListIndex)
        blnDir = (Left(strDatoteka, 1) = "\")
        If blnDir Then
          Form2.strDatoteka = strPot(intStran) & Mid(strDatoteka, 2)
        Else
          Form2.strDatoteka = strPot(intStran) & strDatoteka
        End If
        Form2.blnDir = blnDir
        Form2.Show vbModal, Me
...
End Sub
```

Vsak obrazec projekta je takoj na voljo kjerkoli v programski kodi. Do njegovih gradnikov, javnih spremenljivk in podprogramov imamo dostop posredno, prek sklicevanja na ime obrazca. Za prikaz lastnosti smo obrazcu Form2 sporočili popolno pot do datoteke, določili, ali mapa je ali je ni, in na koncu obrazec prikazali *modalno*. Modalni (*modal*) prikaz obrazca za uporabnika pomeni, da ga bo moral zapreti, preden se lahko vrne in nadaljuje delo, za programerja pa modalni prikaz pomeni začasni prenos izvajanja v kodo drugega obrazca, podoben klicu podprograma. Modalno se prikazujeta tudi okni funkcij MsgBox in InputBox.

Prikaz obrazca dosežemo s klicem metode **Show**. Po želji podamo še dva parametra. Prvi določa modalnost, drugi pa njegovega lastnika. Če parametra izpustimo, se okno prikaže nemodalno.

Kodo bomo lahko preizkusili šele, ko dokončno definiramo kodo drugega obrazca. Začnimo s podatkovnimi strukturami:

```
Option Explicit
Public blnDir As Boolean
Public strDatoteka As String

Private intAtr As Integer
Private intNoviAtr As Integer
```

Za svoje delo potrebuje obrazec dve javni spremenljivki, ki ju nastavljamo iz prvega obrazca, in dve zasebni, ki nam rabita za prikaz in spreminjanje atributov datoteke. Prva procedura naj bo Form_Load. V našem zgledu se pokliče samo enkrat, na začetku izvajanja:

```
Private Sub Form_Load()
  Caption = "Lastnosti"
  Label1.Font.Bold = True
End Sub
```

Podprogram smo izkoristili za nastavitev nekaterih lastnosti, ki bi jih lahko nastavili tudi med načrtovanjem. Programsko kodo se izplača uporabiti pri posebej pomembnih lastnostih, ki jih lahko med načrtovanjem nehote pokvarimo.

Srčika dejavnosti drugega obrazca je odzivna procedura dogodka Activate, ki se pokliče ob vsakem vnovičnem prikazu okna:

```
Private Sub Form_Activate()
Dim chk As CheckBox
Dim lngDolzina As Long
Dim strOpis As String

  ' Izračunaj in izpiši lastnosti...
  Caption = "Lastnosti " & IIf(blnDir, "mape ", "datoteke ")
  Label1.Caption = IIf(blnDir, "Mapa ", "Datoteka ") & _
    IzlusciDatoteko(strDatoteka)

  strOpis = "Pot: " & IzlusciPot(strDatoteka) & vbCrLf

  If Not blnDir Then
    strOpis = strOpis & vbCrLf & "Dolžina: "
    lngDolzina = FileLen(strDatoteka)
    If lngDolzina < 1024 Then
      strOpis = strOpis & Format(lngDolzina, "####") & " B"
    Else
      strOpis = strOpis & Format(FileLen(strDatoteka)/1024, "###,###,###0.0") _
        & " KB"
    End If
  End If
  strOpis = strOpis & vbCrLf & vbCrLf & "Spremenjena: " & _
    Format(FileDateTime(strDatoteka), "dddd, d. mmmm yyyy (hh:mm)")

  Label2.Caption = strOpis

  ' atributi
  intAtr = GetAttr(strDatoteka)
  For Each chk In Check1
    If ((intAtr And chk.Index) > 0) Then
      chk.Value = 1
    Else
      chk.Value = 0
    End If
  Next
  Command2.Enabled = False

  Command1.SetFocus

End Sub
```

Procedura analizira različne parametre datoteke in jih pregledno izpiše. Pri tem nam pomagata že omenjeni funkciji FileLen in FileDateTime ter funkcija za oblikovan izpis, imenovana **Format**. Funkcija Format ima naslednjo obliko:

```
Format(izraz[, oblika[, prviDanTedna[, prviTedenLeta]]])
```

Vrnjena vrednost je vedno niz, ki predstavlja izraz, oblikovan po določilih preostalih parametrov. Njihov pomen je razložen v tabeli:

Parameter	Opis
oblika	Niz, ki določa obliko izpisa. Lahko je opisno ime vnaprej določenih oblik ali simbolični zapis lastne oblike.

Parameter	Opis
prviDanTedna	Ena izmed osmih različnih vrednosti, ki določa dan v tednu, ki v določeni skupnosti velja za prvi dan v tednu. Poleg vsakega posameznega dne v tednu lahko privzamemo tudi sistemsko nastavitev, kar se predpostavi tudi, ko parametra ne zapišemo.
prviTedenLeta	Štiri različne vrednosti, ki določajo merilo, po katerem teden velja za prvi teden v letu. To je lahko teden, v katerem je 1. januar, prvi teden leta, ki vključuje vsaj štiri dni, ali prvi popolni teden v letu. Lahko privzamemo tudi sistemsko nastavitev.

Oglejmo si nekaj poimenovanih oblik (popoln pregled je na voljo v sprotni pomoči):

Ime oblike	Za vrednosti	Opis
Currency	Številske.	Izpis denarne vrednosti, kot je določen v krajevnih nastavitvah.
Standard	Številske.	Običajni zapis realnega števila z dvema decimalnima mestoma.
Percent	Številske.	Realno število, pomnoženo s 100, z dvema decimalnima mestoma in s pripono %. Zapis v odstotkih.
Long Date	Datumske.	Dolga oblika zapisa datuma, kot je določena v krajevnih nastavitvah.
Short Time	Datumske.	Kratek zapis časa v 24-urni obliki, npr. 15:42.

Ker nismo bili zadovoljni s poimenovanimi oblikami, smo v našem zgledu uporabili simbolične zapise lastnih oblik. V teh zapisih znotraj niza uporabimo nekatere posebne simbole, pomembnejši so zbrani v spodnji tabeli, vse pa najdemo v sprotni pomoči:

Simbol	Za vrednosti	Pomen
#	Številske.	Neobvezno številsko mesto.
0	Številske.	Obvezno številsko mesto.
%	Številske.	Zapis v odstotkih.
.	Številske.	Decimalno ločilo (ustrezno krajevnim nastavitvam).
,	Številske.	Ločilo tisočic (ustrezno krajevnim nastavitvam).
d	Datumske.	Zaporedna številka dneva.

Simbol	Za vrednosti	Pomen
dd	Datumske.	Dvomestna zaporedna številka dneva (z morebitno vodilno ničlo).
ddd	Datumske.	Okrajšano ime dneva (ustrezno krajevnim nastavitvam).
dddd	Datumske.	Ime dneva (ustrezno krajevnim nastavitvam).
mmmm	Datumske.	Ime meseca (ustrezno krajevnim nastavitvam).
yy	Datumske.	Letnica brez stoletja.
yyyy	Datumske.	Popolna letnica.
hh	Datumske.	Ura z morebitno vodilno ničlo v 24-urni obliki.

NAMIG Podobne oblike so v rabi tudi v večini namenskih programov. Če ste že uporabljali npr. Microsoft Excel, so vam gotovo znane.

V odzivni proceduri dogodka Activate določimo tudi vklopljenost stikal, ki ponazarjajo atribute datoteke. Zaradi ustrezno nastavljenih zaporednih števil vse uredimo zgolj v eni zanki For Each. Ob nastavitvi stikal onemogočimo stikalo za spreminjanje atributov, saj so sedaj popolnoma ustrezni datotečnim. Zatem še postavimo vnosno točko na privzeti gumb.

Ker je gumb za spreminjanje atributov zdaj onemogočen, moramo poskrbeti za mehanizem, ki ga v ustreznem trenutku znova omogoči. To postori odzivna procedura stikal:

```
Private Sub Check1_Click(Index As Integer)
Dim chk As CheckBox
  intNoviAtr = 0
  For Each chk In Check1
    If chk.Value Then intNoviAtr = intNoviAtr + chk.Index
  Next
  Command2.Enabled = (intNoviAtr <> intAtr)
End Sub
```

Vidimo, da je gumb omogočen le tedaj, ko je nova nastavitev atributov različna od obstoječe.

Za konec moramo obdelati še klika ukaznih gumbov:

```
Private Sub Command1_Click()
  Hide
End Sub

Private Sub Command2_Click()
  SetAttr strDatoteka, intNoviAtr
  Hide
End Sub
```

Privzeti gumb zgolj skrije obrazec, drugi gumb pa pred tem še popravi (zagotovo spremenjene) atribute datoteke. Atribute spremenimo z vgrajeno proceduro **SetAttr**.

S klicem metode **Hide** se modalno prikazano okno skrije in izvajanje se lahko nadaljuje v glavnem obrazcu. Poleg metode Hide imamo za skrivanje obrazca na voljo še stavek Unload, ki ga poznamo že z glavnega obrazca. Ker je glavni obrazec tudi točka, v kateri se začne izvajati celotni program, njegova odstranitev pomeni konec izvajanja. Modalno prikazana okna pa lahko med izvajanjem prikažemo in zatem večkrat odstranimo. Kakšna je razlika med skrivanjem in odstranjevanjem?

Značilnost	Skriti obrazec	Odstranjeni obrazec
Prikaz.	Metoda Show.	Metoda Show.
Skrivanje.	Metoda Hide.	Stavek Unload Me.
Poraba pomnilnika.	Zaseda pomnilnik ves čas izvajanja programa.	Operacijski sistem ga lahko umakne iz pomnilnika, ko je odstranjen.
Hitrost prikaza.	Lahko večja kakor pri odstranjenem obrazcu.	Lahko manjša, če je odstranjen iz pomnilnika.
Dogodek Load	Pripeti se le prvič.	Pripeti se pred vsakim prikazom.

V našem zgledu bi lahko brez posebne škode namesto metode Hide uporabili odstranjevanje s stavkom Unload Me. Navzven posebne razlike ne bo opaziti, saj je obrazec pomnilniško nezahteven in v dogodku Load ne skriva obsežnih obdelav podatkov.

Za konec

Tako, zgled je skoraj končan. Skoraj? Kaj nas še čaka? Hm, znova se je prikradla napaka. Nenadoma je začel prej popolnoma gladko delujoči program povzročati nevšečnost. Do njih pride, kadar pregledujemo lastnost datoteke z desnega seznama. Ko zapremo okno z lastnostmi, se vnosna točka takoj prestavi na levi seznam!

V čem je težava? Spomnimo se obravnave dogodka Activate. Pri projektih z enim obrazcem se dogodek pripeti le pri prvem prikazu obrazca. Če je obrazcev v projektu več, se dogodek Activate lahko pripeti večkrat, kadarkoli obrazec prejme vnosno točko! To lastnost s pridom uporabljamo pri obrazcu Form2, na glavni obrazec pa smo povsem pozabili. Pravilna oblika odzivne procedure dogodka Activate za obrazec Form1 je:

```
Private Sub Form_Activate()
Static blnZnova As Boolean

  If Not blnZnova Then
    blnZnova = True
```

```
    ' Uskladi barvo oznak in določi začetno stran
    intStran = 1
    ZamenjajStran 0

    ' Postavi vnosno točko na prvi seznam
    List1(0).SetFocus

    ' Vklopi upoštevanje seznama pogonov
    blnUpostevajDriveList = True

  End If
End Sub
```

Uporaba statične spremenljivke logičnega tipa je najbolj elegantna rešitev za zgolj enkratno izvajanje določene procedure. Ne pozabimo, da ima statična spremenljivka ob prvem vstopu v podprogram privzeto vrednost (za logični tip je to neresnica), pozneje pa ves čas izvajanja programa ohranja novo prirejeno vsebino. Ko ji vrednost spremenimo v logično resnico, se koda procedure (razen preverjanja pogoja v stavku If) ne bo več izvajala do konca programa.

> **NAMIG** Vso kodo Okenskega komandanta najdete na priloženi plošči CD-ROM, v mapi: \Primeri\2-Delavnica\1-Komandant\5-Koda. Uporaba gradnikov za dostop datotečnega sistema in rekurzivno preiskovanje map je nakazana tudi pri programu WinSeek, ki spremlja visual basic, njena izvorna koda pa je v mapi Samples\Misc\FileCts.

Vaje

Za utrditev znanja in izboljšanje zmogljivosti Okenskega komandanta je tu nekaj nalog:

1. Ukazi za obdelovanje datotek sedaj sprašujejo po potrditvi pred uporabo vsake posamezne datoteke. Uredite ustrezne procedure, tako da se bo vprašanje prikazalo samo za prvo izbrano datoteko, odgovor pa se bo smiselno uporabil za vse druge.
2. Ob izbiri novih pogonov se Okenski komandant vedno postavi v izhodiščno mapo. Popravite program, tako da bo za vsak pogon pomnil izbrano pot in se prestavil vanjo. Npr. če pri izbrani poti C:\Temp\ izberemo pogon D: in zatem znova C:, naj Komandant samodejno izbere pot C:\Temp\. Izboljšamo lahko tudi usklajenost leve in desne strani. Če je npr. levo izbrana pot C:\Program Files\ in uporabnik tudi z desnega seznama izbere pogon C:, naj se samodejno prestavi v isto mapo C:\Program Files\.
3. Oglejte si priloženi zgled WinSeek (glej namig zgoraj). Prenesite njegovo zmožnost preiskovanja map in jo smiselno vgradite v Okenski komandant.

4. Zgled kar kliče po vgraditvi nadziranja napak med izvajanjem. Nehote lahko izberemo nepripravljen pogon; med prepisovanjem lahko zmanjka prostora na disku; datoteka, ki jo želimo odstraniti, je lahko zaščitena... V naslednjem zgledu bomo obdelovanje napak podrobneje spoznali. Preučite torej naslednje poglavje in dogradite nadzor napak še v Okenski komandant.

9. POGLAVJE

Urejevalnik

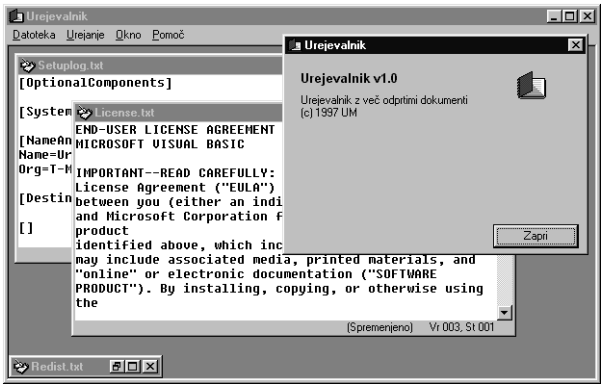
Naš drugi skupni projekt bo razvoj urejevalnika besedil z več odprtimi dokumenti. Po zmogljivostih bo precej podoben Beležki (*Notepad*), le da bo omogočal hkratno obdelavo v več oknih znotraj vmesnika MDI (*Multiple Document Interface*). Naštejmo programerske tehnike, ki jih bomo spoznali ob razvoju:

- Pravila poimenovanja.
- Razvoj programa z vmesnikom MDI.
- Uporaba gradnika ActiveX za standardna pogovorna okna.
- Shranjevanje in odpiranje datotek.
- Delo z odložiščem in predmet Clipboard.
- Nadzor menujev.
- Izdelovanje predlog.
- Obdelava izvajalnih napak in predmet Err.
- Izdelovanje samostojnega programa.
- Izdelovanje vizitke in predmet App.
- Branje parametrov ukazne vrstice.
- Uporaba funkcij Windows API.
- Čarovnik za pripravo programov.

Kakšen je delujoči program, vidimo na sliki 9-1. Izvorna koda zgledov je na priloženi plošči CD-ROM, v mapah znotraj mape \Primeri\2-Delavnica\2-Urejevalnik\.

Pravila poimenovanja

Naš prvi projekt, Okenski komandant, je imel razmeroma preprost uporabniški vmesnik. Pri uporabi gradnikov smo zato puščali privzeta imena, kot sta Label1 ali Form2. Podrobnejši ogled izvorne kode pa razkrije, da smo bili pri poimenovanju spremenljivk bolj zgovorni, saj smo vsako označili z ustrezno predpono.



Slika 9-1: Urejevalnik z vmesnikom MDI

Razvojno okolje visual basica pesti velika slabost, ki se je zlepa ne more znebiti. Če kak gradnik že uporabimo v programski kodi in mu naknadno spremenimo ime, postane programska koda v najboljšem primeru neuporabna (mrtva koda), večinoma pa celo nevarno leglo napak med izvajanjem. Temu bolj učeno pravimo enosmerna narava vizualnih orodij. Pri načrtovanju vmesnika se moramo zato zavedati naslednjega pravila:

Če bomo gradnikom spreminjali ime, storimo to, preden zapišemo eno samo vrstico kode.

S tem pravilom v mislih si oglejmo neformalne napotke za poimenovanje gradnikov in spremenljivk (*naming conventions*). Neformalne zato, ker nam njihove rabe nihče ne vsiljuje in niso nikjer natančno določeni. Pravzaprav ni tako pomembno, kakšna pravila poimenovanja uporabljamo, pomembneje je, da se jih dosledno držimo v vseh naših projektih. S tem izboljšamo berljivost kode in si izdatno olajšamo poznejše vzdrževanje.

Navodila, ki spremljajo visual basic, predlagajo naslednja pravila poimenovanja:

- Vsaki spremenljivki dodelimo ime, katerega predpona odraža njen doseg in podatkovni tip.
- Vsakemu gradniku dodelimo ime, katerega predpona določa njegovo vrsto.

Oglejmo si krajši povzetek predpon:

Predpona	Določa
g	Globalni doseg.
m	Modulski doseg.
(brez)	Podprogramski doseg.
bln	Tip Boolean.

Predpona	Določa
int	Tip Integer
str	Tip String
vnt	Tip Variant
frm	obrazec (gradnik *Form*)
chk	stikalo (gradnik *CheckBox*)
mnu	menujska izbira (gradnik *Menu*)
cmd	gumb (gradnik *CommandButton*)
ctr	poljuben gradnik (predmet *Control*)

NAMIG Popoln pregled pravil poimenovanja najdete v sprotni pomoči (**Help ➪ Books Online**, iščite *Naming Conventions*).

Nekaj zgledov napovedi:

```
Public gintStevilo     ' globalno dosegljivo celo število
...
Private mstrBesedilo   ' modulsko dosegljiv niz
...
Dim cmdGumb            ' ukazni gumb
```

V kodnih zgledih, ki jih bomo razvili v nadaljevanju knjige, bomo upoštevali predpone za vrsto, ne pa tudi predpone dosegljivosti. Kot že rečeno, je pomembneje, da se pravil dosledno držimo, kakor to, kako natančna so. Vsekakor naj nam pomagajo pri branju in vzdrževanju programske kode.

Uporabniški vmesnik

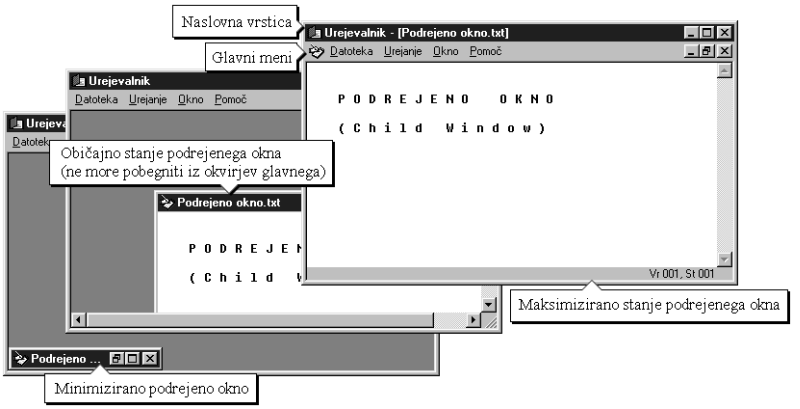
Urejevalnik uporablja uporabniški vmesnik, ki dopušča več hkrati odprtih dokumentov. Podobne vmesnike uporablja večina namenskih programov za obdelovanje dokumentov (npr. Microsoft Word, Corel DRAW!, Lotus 1-2-3 itn.). Oglejmo si, kako ga pripravimo z visual basicom.

NAMIG Za pripravo vmesnika MDI, pa tudi SDI (*Single Document Interface*) in vmesnika, ki posnema *Raziskovalca* (*Explorer*), je na voljo poseben pripomoček – *čarovnik*. Čarovnik (*wizard*) je zaporedje pogovornih oken, ki nas korak za korakom vodi po bolj zapletenih operacijah. Ker je namen naslednjega zgleda spoznavati podrobnosti

samega vmesnika, ga bomo razvili ročno. Pri naslednjih projektih lahko brez zadržkov uporabimo čarovnik, ki ga bomo podrobneje spoznali ob koncu tega poglavja.

Ko v visual basicu začnemo nov projekt (Standard EXE), nas pričaka prazen obrazec Form1. Tega bomo spremenili v *podrejeno* okno vmesnika MDI. Podrejeno okno (*child window*) je okno v glavnem oknu. Če ga pomanjšamo, se ne pospravi v opravilno vrstico (*taskbar*), ampak je ponazorjeno z ikono pri dnu glavnega okna. Ko ga povečamo, se njegova naslovna vrstica zlije z naslovno vrstico glavnega okna, nadzorni menu pa z glavnim menujem. Ta različna stanja podrejenega okna prikazuje naslednja slika:

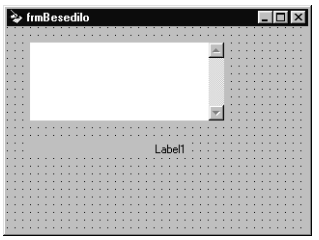
Slika 9-2: Stanja podrejenega okna



Pri vmesniku MDI je podrejenih oken lahko več, glavno pa je vedno le eno. Samo to je ponazorjeno z ustreznim gumbom v opravilni vrstici. S posebnimi ukazi lahko podrejena okna urejamo na različne načine.

Kako bomo pripravili podrejeni obrazec? Najprej si oglejmo sliko uporabniškega vmesnika:

Slika 9-3: Uporabniški vmesnik podrejenega okna



Na obrazec namestimo vnosno polje in oznako. Njun položaj sploh ni pomemben, nastaviti pa moramo naslednje lastnosti:

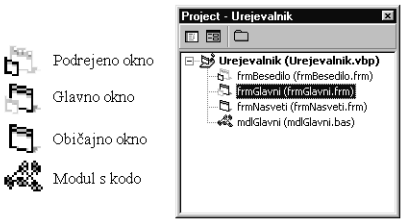
Gradnik	Lastnost	Nastavitev
obrazec (*Form*)	(Name)	frmBesedilo
	MDIChild	True
	BorderStyle	2 - Sizeable
	Icon	(datoteka frmBesedilo.ico)
vnosno polje (*TextBox*)	(Name)	txtBesedilo
	BorderStyle	0 - None
	Font	(Fixedsys, 9)
	MultiLine	True
	ScrollBars	2 - Vertical
	Text	(prazen niz)
oznaka (*Label*)	(Name)	lblStatus
	Alignment	1 - Right Justify

Pazimo, da gradnike (lastnost Name) pravilno poimenujemo. Na ta imena se bomo sklicevali iz programske kode. Ikona obrazca je lahko katerakoli datoteka .ICO, omenjeno pa je moč najti na priloženi plošči CD-ROM.

Ko smo obrazcu frmBesedilo nastavili lastnost **MDIChild** na resnično vrednost, smo ga s tem določili za podrejeni obrazec vmesnika MDI. Kako pa pridemo do glavnega obrazca?

Glavni obrazec vmesnika MDI dodamo v projekt z ukazom **Add MDIForm** iz glavnega menuja **Project**. Če se prikaže galerija predlog, pustimo izbran osnovni obrazec, ki ga odpremo z gumbom Open. Glavni obrazec se že na prvi pogled razlikuje od podrejenega. Na vrsto obrazca oziroma modula nas opozarja tudi njegova ikona v projektnem oknu. Različne ikone vidimo na naslednji sliki:

Slika 9-4: Ponazoritev vrste modula v projektnem oknu



Na glavno okno vmesnika MDI lahko *neposredno* izrišemo le nevidne gradnike (npr. uro, *Timer*) in izbor gradnikov, ki poznajo lastnost **Align**. Od vgrajenih je tak le slikovni okvir (*PictureBox*), ki ga bomo natančneje spoznali v naslednjem poglavju. Na srečo je slikovni okvir vsebnik, tako da lahko pozneje vanj izrišemo poljuben gradnik. Tako lahko določimo vrstice z gumbi in vrstico stanja.

Za potrebe urejevalnika bomo glavnemu oknu MDIForm1 nastavili nekaj lastnosti. Povzete so v spodnji tabeli:

Gradnik	Lastnost	Nastavitev
Obrazec MDI (*MDIForm*)	(Name)	frmGlavni
	AutoShowChildren	False
	BorderStyle	2 - Sizeable
	ScrollBars	True
	Caption	"Urejevalnik MDI"
	Icon	(datoteka frmGlavni.ico)
	StartUpPosition	3 - Windows Default

Ne pozabimo na nastavitev lastnosti Name! Za dokončno oblikovanje vmesnika bomo na glavnem oknu določili menujsko vrstico. Pred tem mu bomo dodali še nov gradnik ActiveX, ki omogoča dostop do standardnih pogovornih oken.

Dodajanje gradnikov ActiveX

Zmožnosti visual basica lahko praktično neomejeno širimo z dodatnimi gradniki ActiveX[30]. Ti so pravzaprav posebni programi, navedeni v registru (*registry*), ki se vedejo kakor strežniki[31]. Njihova raba ni omejena samo na visual basic, ampak jih lahko uporabljamo v najrazličnejših razvojnih okoljih (npr. Borland Delphi) in tudi namenskih programih (npr. Microsoft Internet Explorer). Količina gradnikov ActiveX, ki jih imamo na voljo, je odvisna od različice visual basica, ki jo uporabljamo, ter od drugih programov, nameščenih v naš računalnik. Pregled zunanjih gradnikov, ki jih lahko vključimo v projekt, dobimo z ukazom **Components** iz glavnega menuja **Project**.

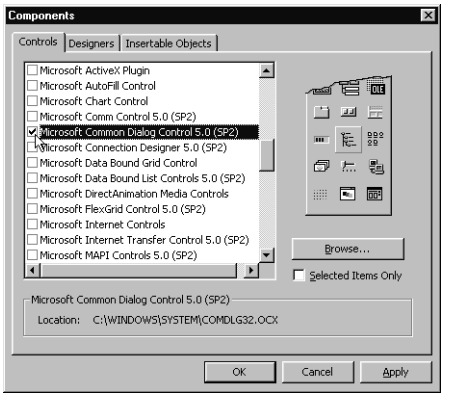
Za potrebe Urejevalnika bomo uporabili en sam dodaten gradnik, to je Microsoft Common Dialog Control. Z njim imamo dostop do standardnih pogovornih oken za odpiranje in shranjevanje datotek, izbor tiskalnika, izbor pisave ter še nekaterih. V projekt ga vključimo tako, da vklopimo stikalo v njegovi vrstici (glej sliko). Ko potrdimo vnos in zapremo okno Components, bo novi gradnik ponazorjen z novo ikono v orodjarni (slika 9-6).

> **NAMIG** Skupaj z novim gradnikom je v razvojnem okolju postala dostopna tudi sprotna pomoč, ki olajša njegovo rabo.

[30] V 16-bitnih različicah visual basica so bili zunanji gradniki označeni z VBX. Sledili so jim 32-bitni gradniki OLE, označeni z OCX. Ti so dobili novo tržno ime, gradniki *ActiveX*. (črka X, ki ves čas vztraja, je kratica za *eXtension* – razširitev, kar gradniki pravzaprav so).

[31] Strežnikom, odjemalcem in protokolu ActiveX je namenjen četrti del knjige.

Slika 9-5: Izbor razpoložljivih gradnikov ActiveX



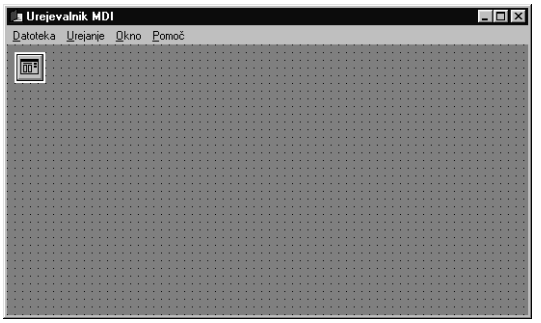
Slika 9-6: Nov gradnik v orodjarni

Raba zunanjih gradnikov ActiveX lahko najbolj zapletena programerska opravila poenostavi v nastavitev nekaj lastnosti ali klic nekaj metod. Cena, ki jo plačamo za to udobje, je najprej cena samega gradnika, potem dodaten prostor na disku, ki ga zaseda datoteka .OCX[32], in težave pri prenašanju programa v drug računalnik, saj moramo skupaj s programom namestiti tudi ustrezno različico vseh uporabljenih zunanjih gradnikov.

Sedaj imamo na voljo vse potrebne gradnike, da dokončamo uporabniški vmesnik glavnega obrazca. Izdelali ga bomo po zgledu s slike 9-7.

Gradnik CommonDialog je neviden, zato ga lahko izrišemo neposredno na površino glavnega okna. Njegove lastnosti bomo večinoma nastavili iz programske kode, med načrtovanjem pa moramo poskrbeti za njegovo opisno ime:

[32] Ta sploh ni zanemarljiv. Spomnimo se namestitve visual basica in prostora, ki ga zahteva v sistemski mapi.



Slika 9-7: Glavni obrazec med oblikovanjem

Gradnik	Lastnost	Nastavitev
Standardni pogovor (*CommonDialog*)	(Name)	dlgCommonDialog

Kakor vidimo, je priporočena predpona za ta gradnik kar »dlg«.

Načrtovanje menujev

Pri vmesniku MDI prikaže menu samo glavno okno. Določimo ga lahko tudi za podrejena okna, vendar po zagonu programa ne bo v njih, temveč bo menu podrejenega okna *nadomestil* glavni menu. Pri načrtovanju menujev imamo torej dve možnosti:

- Pripravimo glavni menu, v katerem ni ukazov za delo z vsebino dokumenta. Zatem vse ukaze glavnega menuja ponovimo v menuju podrejenega okna, ki mu dodamo še nove ukaze za delo z vsebino.
- Pripravimo samo glavni menu, z vsemi ukazi. Vidnost ukazov za delo z vsebino dokumenta nadziramo s programsko kodo.

Druga možnost je verjetno privlačnejša, saj je definiranje menujev precej zamudno, po drugi strani pa se v tem delu knjige učimo programiranja v visual basicu, zato ne bo odveč, če si ogledamo možnosti nadzora menujev iz programske kode.

V spodnji tabeli je povzetek definiranih menujev za glavno okno:

Ime menuja	Ime izbire[(zap. št.)]	Napis	Bližnjica	Dodatno
mnuDatoteka		"&Datoteka"		
	mnuDatotekaNova	"&Nova"	Ctrl+N	
	mnuDatotekaOdpri	"&Odpri..."	Ctrl+O	
	mnuDatotekaZapri	"&Zapri"		
	mnuLocilo1	"-"		
	mnuDatotekaShrani	"&Shrani"	Ctrl+S	

Ime menuja	Ime izbire[(zap. št.)]	Napis	Bližnjica	Dodatno
	mnuDatotekaShraniKot	"Shr&ani kot..."		
	mnuLocilo2	"-"		
	mnuDatotekaKonec	"&Konec dela"		
mnuUrejanje		"&Urejanje"		
	mnuUrejanjeRazveljavi	"&Razveljavi"		
	mnuLocilo3	"-"		
	mnuUrejanjeUkaz(1)	"&Izreži"	Ctrl+X	
	mnuUrejanjeUkaz(2)	"&Kopiraj"	Ctrl+C	
	mnuUrejanjeUkaz(3)	"&Prilepi"	Ctrl+V	
	mnuLocilo4	"-"		
	mnuUrejanjeIsci	"&Išči..."	Ctrl+F	
	mnuUrejanjeZnova	"Išči &znova"	F3	
mnuOkno		"&Okno"		WindowList = True
	mnuOknoUredi(1)	"Razporedi &vodoravno"		
	mnuOknoUredi(2)	"Razporedi &navpično"		
	mnuOknoUredi(3)	"Razporedi v &kaskado"		
	mnuOknoUredi(4)	"Uredi &ikone"		
mnuPomoc		"&Pomoč"		
	mnuPomocNasveti	"&Nasveti"	F1	

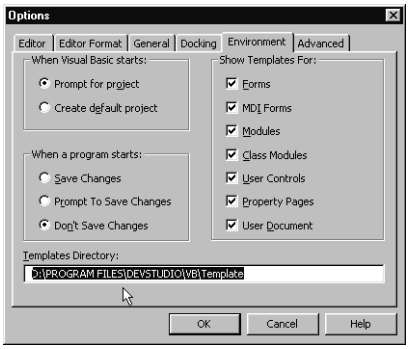
Pazimo: menujske izbire, ki sodijo v polje gradnikov, so označene z imenom in pripadajočo zaporedno številko (lastnostjo Index) v oklepaju! Značilnost glavnega menuja pri vmesniku MDI je menu Okno (*Window*), v katerem so ukazi za urejanje podrejenih oken. Temu menuju nastavimo lastnost **WindowList** na resnično vrednost. Po zagonu programa se bo na dno tega menuja samodejno dodala nova izbira za vsako odprto podrejeno okno.

Tako pripravljen menu nam je naložil precej dela (seveda tistim, ki ste ga pripravljali ročno in niste posegli po priloženi plošči CD-ROM :-). Vidimo, da je uporaben v večini projektov, ki obdelujejo neko vsebino znotraj vmesnika MDI. Bi bilo mogoče vloženi trud hitro uporabiti tudi v drugih projektih?

Predloge

Visual basic nam omogoča pripravo *predlog*. Predloga (*template*) ni nič drugega kakor obrazec ali modul s kodo, shranjen v posebni mapi. Pot do te mape se skriva

v nastavitvah razvojnega okolja. Najdemo jo lahko v pogovornem oknu ukaza **Tools ➪ Options**, pod jezičkom **Environment** v vnosnem polju **Templates Directory**.



Slika 9-8: Mapa shranjenih predlog

Ta mapa vsebuje nekaj podmap. Nekatere so naštete v spodnji tabeli:

Podmapa	Vsebina
Forms	Predloge obrazcev.
MDIForms	Predloge obrazcev MDI.
Modules	Predloge modulov s kodo.
Projects	Predloge projektov.

Kot predloge večinoma shranjujemo delno dokončane obrazce, ki jih lahko malo prikrojene ali dopolnjene uporabimo v več projektih. To, kar smo doslej postorili z glavnim oknom, bi se vsekakor splačalo shraniti kot predlogo. Kadarkoli bomo želeli v projekt dodati glavno okno MDI, se bo odprla galerija pripravljenih predlog in med njimi bo tudi naše glavno okno z menujem.

Ves postopek se glasi:

1. Izberemo ukaz **File ➪ Save frmGlavni As...**
2. Izberemo mapo predlog in v njej podmapo »MDIForms«.
3. V vnosno polje File name vnesemo dovolj zgovorno opisno ime predloge, npr. »MDI form z menujem«.
4. Izberemo gumb **Save**.

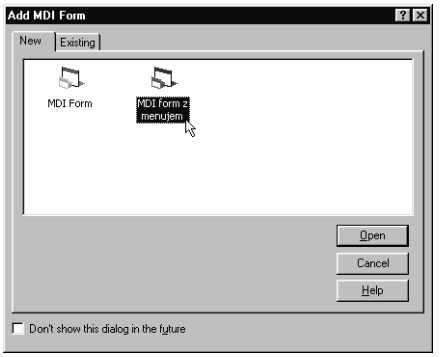
Pripravljen obrazec, shranjen kot predlogo, bomo sedaj zavrgli in ga na novo odprli iz predloge.

5. V glavnem menuju **Project** izberemo ukaz **Remove frmGlavni**.
6. Na vprašanje o shranjevanju odgovorimo negativno, kliknemo gumb **No**.

Zdaj bomo odprli nov obrazec MDI, ki bo temeljil na prej pripravljeni predlogi.

7. V glavnem menuju **Project** izberemo ukaz **Add MDI Form**.
8. Prikaže se galerija, v kateri je tudi naša predloga.
9. Izberemo predlogo »MDI form z menujem«.
10. Odpremo nov obrazec, temelječ na predlogi, kliknemo gumb **Open**.

Slika 9-9: Galerija pripravljenih predlog je le odsev datotek ustrezne mape.



Ko bomo novi obrazec želeli shraniti, nas bo visual basic vprašal po imenu datoteke. Seveda, saj ne shranjujemo spremenjene predloge, temveč obrazec, ki le *temelji* na njej. Tako kakor vse druge obrazce, ga moramo prvič shraniti pod določenim imenom. Če želimo spremeniti samo predlogo, moramo posodobljen obrazec znova shraniti v prej omenjeno posebno mapo.

Predloga lahko poleg uporabniškega vmesnika vsebuje tudi programsko kodo. Vse to podedujejo vsi obrazci, ki temeljijo na njej. Pozneje bomo razvili zgled samostojnega okna z nekaj programske kode, ki nam lahko rabi kot predloga. Predloga je lahko tudi ves projekt. Shranimo ga v podmapo »Projects«.

NAMIG Programsko kodo, razvito do te točke, najdete na priloženi plošči CD-ROM v mapi \Primeri\2-Delavnica\2-Urejevalnik\1-Vmesnik\.

Standardna pogovorna okna

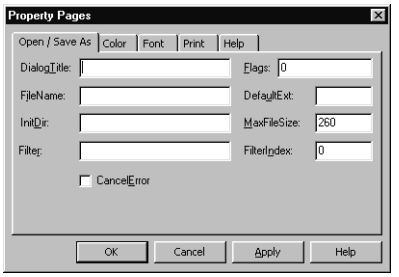
Standardna pogovorna okna so vpeljali v operacijski sistem v želji, da bi čim več programov uporabljalo podoben dostop do poglavitnih sistemskih sredstev, kot so datoteke, tiskalniki, razpoložljive pisave, barve in podobno. S prihodom Oken 95 so postala standardna pogovorna okna za odpiranje in shranjevanje datotek tako

zmogljiva, da resnično ni več smiselno razvijati lastnih nadomestkov. Prednost standardnih pogovornih oken je tudi v tem, da so že prilagojena krajevnim nastavitvam. Tako se bo enak program v različnih sistemih samodejno prilagodil npr. uporabljenemu jeziku.

Uporaba vseh standardnih pogovornih oken postane mogoča z vključitvijo zunanjega gradnika Microsoft Common Dialog Control. Njegove lastnosti in metode omogočajo izbiro vrste pogovornega okna in njegov prikaz. Oglejmo si zanimivejše lastnosti in metode.

Lastnosti

Večina zunanjih gradnikov ActiveX, med katere sodi tudi obravnavani gradnik, ponudi v oknu za nastavljanje lastnosti še posebno nastavitev, imenovano (Custom). Ta je pravzaprav pripomoček (imenovan *Property Pages*), ki prikaže posebno pogovorno okno, v katerem lahko pregledno nastavljamo več lastnosti, ki so smiselno razdeljene v skupine. Vse te lastnosti so na voljo tudi neposredno, vendar jih je s tem pripomočkom laže uporabljati.



Slika 9-10: Pripomoček za pregledno nastavljanje lastnosti standardnega pogovornega okna

Ker isti gradnik skrbi za več različnih standardnih pogovornih oken, ima precej lastnosti, ki so značilne za posamezno podvrsto okna. Mi se bomo osredotočili predvsem na lastnosti, ki vplivajo na prikaz oken za odpiranje in shranjevanje datotek, ki jih tudi najpogosteje uporabljamo. Podroben opis gradnika najdemo v sprotni pomoči (izberemo gradnik na vmesniku in pritisnemo tipko F1).

Lastnost **DialogTitle**

Lastnost določi niz, ki se izpiše v naslovni vrstici pogovornega okna.

Lastnost **CancelError**

Če lastnost CancelError nastavimo na resnično vrednost, bo izbira gumba za preklic povzročila napako med izvajanjem, ki jo lahko prestrežemo in ustrezno obdelamo.

Lastnost **DefaultExt**

Lastnost določa privzeti podaljšek, ki se pri shranjevanju samodejno doda datotekam, poimenovanim brez podaljška. Imeti mora tudi piko, npr. ».TXT«.

Lastnosti **Filter** in **FilterIndex**

Pri obravnavanju gradnika seznam datotek (*FileListBox*) smo z lastnostjo Pattern določali vzorec imena datotek, ki so se prikazale na seznamu. Podobno vlogo pri standardnih pogovornih oknih opravljata lastnosti Filter in FilterIndex. Lastnost Filter je niz, ki določa več parov opisnega imena vzorca in njegove dejanske vrednosti. Elementa para ločimo z navpično črto (znak |), pare med sabo pa s podpičjem (;). Lastnost FilterIndex določa, kateri izmed parov je na začetku izbran; prvemu pripada zaporedna številka 1. Ker je to privzeta nastavitev, lastnosti dostikrat niti ne nastavljamo posebej. Zgled:

```
dlgCommonDialog.Filter = _
  "Besedilo |*.txt;*.doc;*.ini|Vse datoteke (*.*)|*.*"
```

Lastnost **InitDir**

Lastnost določa izhodiščno mapo, ki jo na začetku prikaže pogovorno okno za odpiranje ali shranjevanje datotek.

Lastnost **Flags**

Lastnost Flags omogoča nastavitev cele gore parametrov z vnaprej pripravljenimi konstantami, ki jih lahko kombiniramo s seštevanjem ali bitnim operatorjem Or. Nekaj zanimivejših vrednosti je zbranih v naslednji tabeli:

Vrednost	Pomen
cdlOFNAllowMultiselect	Dopušča izbor več datotek.
cdlOFNCreatePrompt	Če uporabnik izbere neobstoječo datoteko, ga pogovorno okno vpraša, ali želi ustvariti novo.
cdlOFNFileMustExist	Zahteva, da uporabnik izbere obstoječo datoteko. Zelo uporabno pri odpiranju datotek.
cdlOFNHideReadOnly	Skrije stikalo Read Only.
cdlOFNNoDereferenceLinks	Ne uporablja bližnjic (*shortcuts*), ampak jih prikazuje kot navadne datoteke.
cdlOFNNoLongNames	Ne uporablja dolgih imen datotek.
cdlOFNNoReadOnlyReturn	Ne dopušča izbire datotek, zaščitenih proti pisanju (z nastavljenim atributom Read Only).
cdlOFNOverwritePrompt	Zahteva potrditev pri izbiri obstoječe datoteke (uporabno samo pri shranjevanju).
cdlOFNPathMustExist	Zahteva izbiro obstoječe poti (ta vrednost se samodejno predpostavi pri cdlOFNFileMustExist).

Lastnosti **FileName** in **FileTitle**

Lastnosti sta dostopni le med izvajanjem programa. Prva določa popolno pot do izbrane datoteke, druga pa le ime datoteke, brez poti. Lastnost FileName lahko uporabimo za nastavitev privzetega imena datoteke.

Metode

Metode **Show...**

Metode, katerih ime se začne s Show, prikažejo različna standardna pogovorna okna modalno. Naštete so v spodnji tabeli:

Metoda	Prikaže standardno pogovorno okno
ShowOpen	Okno za odpiranje datotek.
ShowSave	Okno za shranjevanje datotek.
ShowColor	Okno za izbiranje barve.
ShowFont	Okno za izbiranje pisave.
ShowPrinter	Okno za izbiranje tiskalnika ali okno za nastavitev tiskalnika.
ShowHelp	Okno s sprotno pomočjo.

Pri Urejevalniku bomo standardna pogovorna okna uporabili za izbiranje datotek, torej bomo uporabili metodi ShowOpen in ShowSave.

Programska koda

Tudi v našem drugem zgledu bomo programsko kodo razvili v nekaj korakih, ki bodo osvetlili nekatere pomembnejše programerske tehnike v visual basicu. Pri delu z vmesnikom MDI je najpomembneje določiti programski modul, v katerega bomo uvrstili določene podprograme. Zaradi čim večje modularnosti programa je dobro, da podrejeno okno kar največ postori samo. Tako ga bomo zlahka lahko uporabili še v drugih projektih. Prenos čimveč opravil v podrejeno okno ima še eno prednost: glavno okno bo ostalo dovolj splošno, da ga bomo lahko uporabili tudi v drugih projektih, npr. v urejevalniku slik.

Po drugi strani nadzor celotnega programa iz podrejenega okna ni mogoč, saj se podrejena okna odpirajo in zapirajo, in je prav mogoče, da v določenem trenutku ni odprto nobeno. Urejevalnik bomo zato nadzirali iz podprogramov glavnega okna. Glavno okno je seveda tudi križišče pri izbiri različnih menujskih ukazov.

Glavno okno

Glavno okno moramo ob zagonu programa vzpostaviti. Pri vmesniku MDI je najbolje, da glavno okno vzpostavimo, kakor navadno, znotraj dogodka Load:

```
Private Sub MDIForm_Load()
  ' Določimo stalne nastavitve standardnih pogovorov
  With dlgCommonDialog
    .Filter = "Besedilo (*.txt)|*.txt|Vse datoteke (*.*)|*.*"
    .DefaultExt = ".txt"
  End With
  ' Vzpostavimo menuje
  OmogociMenuje
End Sub
```

Najprej smo poskrbeli za lastnosti gradnika dlgCommonDialog, ki skrbi za prikaz standardnih pogovornih oken. Nastavili smo mu dve pomembni lastnosti, ki sta sicer na voljo tudi med načrtovanjem. Prednosti nastavljanja določenih lastnosti v programski kodi smo že omenili pri Okenskem komandantu. Na koncu še vzpostavimo menuje. To postori nova, lastna javna procedura OmogociMenuje:

```
' Javna procedura, ki določi vidnost oziroma
' omogočenost nekaterih menujskih izbir.
Public Sub OmogociMenuje()
Dim blnObstajaOkno As Boolean
  blnObstajaOkno = Not (ActiveForm Is Nothing)
  mnuDatotekaZapri.Enabled = blnObstajaOkno
  mnuDatotekaShrani.Enabled = blnObstajaOkno
  mnuDatotekaShraniKot.Enabled = blnObstajaOkno
  mnuUrejanje.Visible = blnObstajaOkno
  mnuOkno.Visible = blnObstajaOkno
End Sub
```

Procedura je javna in je s tem postala nova metoda glavnega okna, dostopna iz vseh drugih modulov. Njena naloga je, da v menujih odrazi stanje programa. Edini vzrok spreminjanja menujev je navzočnost podrejenega okna. Vsi glavni obrazci MDI poznajo lastnost **ActiveForm**. Lastnost kaže na trenutno dejavni podrejeni obrazec. Izraz

```
ActiveForm Is Nothing
```

je resničen natančno tedaj, ko je glavno okno prazno. Če to drži, onemogočimo ukaze za shranjevanje in zapiranje podrejenega okna ter skrijemo celotne menuje za urejanje in nadziranje oken. To javno proceduro bomo morali poklicati vedno, ko spremenimo število odprtih podrejenih oken.

Zdaj določimo še odzivno proceduro za konec dela:

```
' Konec dela
Private Sub mnuDatotekaKonec_Click()
  Unload Me
End Sub
```

Pri vmesniku MDI je še kako pomembno, da glavno okno zapremo pravilno. Kakor bomo videli v nadaljevanju, nam uporaba stavka Unload omogoči udoben in varen konec dela, ne glede na število odprtih podrejenih oken.

Program lahko sedaj že preizkusimo. Opazili bomo, da je menujska vrstica brez podrejenih oken precej pusta.

Slika 9-11: Videz glavnega okna brez podrejenih oken

NAMIG Pri glavnem oknu vmesnika MDI imamo na voljo lastnost **AutoShowChildren**. Če je nastavljena na resnično vrednost, bo glavno okno že ob zagonu samodejno ustvarilo eno podrejeno okno. Na splošno se bolj izplača lastnost nastaviti na neresnično vrednost, kakor v našem zgledu, in prikaz prvega okna nadzirati programsko, znotraj metode Load.

Podrejeno okno

Pri razvoju podrejenega okna se moramo ves čas zavedati, da ne razvijamo enega samega okna, temveč *vzorec*, po katerem bo lahko nastal cel kup podrejenih oken.

Vzpostavitev

Njegovo vzpostavitev lahko zapišemo znotraj odziva na dogodek Load, vendar je še primernejši dogodek Initialize, ki se pripeti še prej.

NAMIG Podrejene obrazce lahko uporabimo v več *izvodih*. Čeprav posamezne izvode odstranjujemo s stavkom Unload, bo njihovo vzpostavljeno stanje, določeno v odzivu na dogodek Initialize, vztrajalo v pomnilniku, dokler je na voljo še kak izvod.

Pred vzpostavitvijo določimo podatkovne strukture, ki jih potrebujemo pri delu:

```
Option Explicit
' Dodatni lastnosti obrazca:
```

```
Public blnNeimenovan As Boolean  ' ali ima okno pravilno ime?
Public strDatoteka As String     ' ime okna, če je poimenovano
' Zaščiteno
Private blnSprememba As Boolean  ' označuje spremembe besedila
```

Obrazcu smo dodali dve javni spremenljivki in s tem pravzaprav določili dve novi *lastnosti*. Dostopni bosta kjerkoli v programski kodi. Zasebna spremenljivka bo rabila le za implementacijo podprogramov znotraj obrazca.

Ob vzpostavitvi je pravilno, da poskrbimo za začetne vrednosti vseh modulskih in globalnih spremenljivk:

```
' Lepa programerska praksa je, da poskrbimo
' za natančno določene začetne vrednosti
Private Sub Form_Initialize()
  blnSprememba = False
  blnNeimenovan = True
  strDatoteka = ""
End Sub
```

Čeprav bi se lahko zanašali na privzete vrednosti (razen za spremenljivko `blnNeimenovan`), je z eksplicitno zapisanimi stavki koda razumljivejša in bolj trdoživa.

Dogodku Initalize je nasproten dogodek Terminate. Ko se pripeti, je obrazec pred tem, da se dokončno odstrani iz pomnilnika. Zato se moramo v njem ustrezno odzvati:

```
' Lahko da se je zaprlo zadnje okno
Private Sub Form_Terminate()
  frmGlavni.OmogociMenuje
End Sub
```

S klicem javne metode glavnega obrazca zagotovimo usklajenost menujev s številom prikazanih podrejenih obrazcev.

Pri podrejenem oknu moramo poskrbeti še za eno pomembno opravilo. Spomnimo se načrtovanja. Pri risanju gradnikov smo dejali, da njun položaj ni pomemben. Zakaj? Ker ima podrejeno okno okvir spremenljive velikost (lastnost BorderStyle), bo morda treba večkrat preračunati velikost in položaj gradnikov. Prava procedura za tako delo je odziv na dogodek Resize. Ker se dogodek zagotovo pripeti vsaj enkrat tudi pri prvem prikazu okna, je položaj gradnikov med načrtovanjem nepomemben. Procedura je videti takole:

```
' Sprememba velikosti okna
Private Sub Form_Resize()
  ' Zagotovimo, da sta vnosno polje
  ' in vrstica stanja vedno
  ' raztegnjena čez ves obrazec
  If ScaleHeight > lblStatus.Height Then
    lblStatus.Move 0, ScaleHeight - lblStatus.Height, ScaleWidth
    txtBesedilo.Move 0, 0, ScaleWidth, ScaleHeight - lblStatus.Height
  End If
End Sub
```

Vse delo postori metoda Move, ki smo jo že spoznali pri obravnavanju gradnikov. Vidimo, da trenutno širino obrazca ugotovimo iz lastnosti ScaleWidth, višino pa iz lastnosti ScaleHeight. Zakaj nista primerni kar Width in Height? Slednji dve lastnosti opisujeta celotno širino in višino obrazca, skupaj z obrobo, naslovno vrstico, morebitnim menujem itn. ScaleWidth in ScaleHeight pa opisujeta zgolj *delovno površino* obrazca (*client area*). Z njima se bomo natančneje spoznali, ko bomo obravnavali grafiko. Ker je negativna velikost gradnika prepovedana, moramo s stavkom If zagotoviti, da izračunana višina gradnika ne bo nepravilna. To se lahko kaj hitro zgodi, npr. ko podrejeno okno pomanjšamo.

Preden bomo lahko preverili vedenje podrejenega okna, moramo v glavnem oknu določiti mehanizem njihovega odpiranja in zapiranja.

Nadzor

Nadzor ustvarjanja in zapiranja podrejenega okna moramo določiti v glavnem oknu. Za to smo si že pripravili ustrezna menujska ukaza. Dodajmo še pripadajoči odzivni proceduri:

```
' Odprem novo okno
Private Sub mnuDatotekaNova_Click()
  NovoOkno
  OmogociMenuje
End Sub

' Zaprem dejavno okno
Private Sub mnuDatotekaZapri_Click()
  Unload ActiveForm
  OmogociMenuje
End Sub
```

Za odpiranje novega okna bo skrbela lastna procedura, zapiranje pa je stvar preprostega stavka Unload, ki mu kot parameter posredujemo dejavno okno (lastnost ActiveForm). V obeh primerih moramo obnoviti stanje menujev.

Začetna oblika procedure NovoOkno je zelo preprosta:

```
Private Sub NovoOkno()
Dim frm As frmBesedilo
  Set frm = New frmBesedilo
  frm.Show
End Sub
```

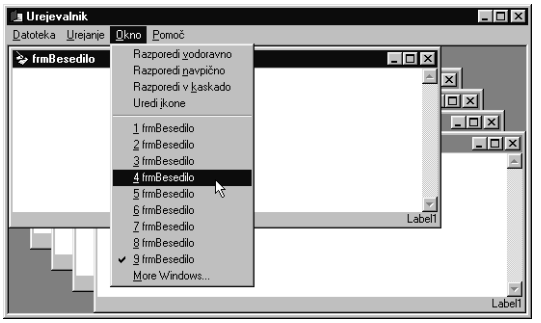
Izdelovanje novih izvodov poznamo že iz prvega dela (obravnava predmetov), tako da bomo le na hitro obnovili znanje. Spremenljivka `frm` je predmetna spremenljivka, ki kaže na prototip podrejenega okna, določenega z obrazcem frmBesedilo. V stavku Set spremenljivko uporabimo z implicitno stvaritvijo, kar pomeni, da se bo novi izvod obrazca samodejno ustvaril, brž ko ga naslovimo, npr. z metodo Show, ki še poskrbi za njegov prikaz.

NAMIG Programsko kodo, razvito do te točke, najdete na priloženi plošči CD-ROM v mapi \Primeri\2-Delavnica\2-Urejevalnik\2-Koda\.

Urejanje podrejenih oken

Zgled je zdaj že toliko razvit, da lahko v njem odpremo poljubno število podrejenih oken. Opazujmo, kaj se dogaja z menujem Okno. Zaradi nastavljene lastnosti WindowList je v njem poleg vnaprej določenih ukazov vedno na voljo tudi izbira za vsako podrejeno okno. Izbire niso preveč zgovorne, menujska izbira namreč odseva napis na oknu, kot ga določa lastnost Caption.

Slika 9-12: V menuju Okno je vedno posodobljen seznam odprtih podrejenih oken.



Poskrbimo še za obravnavo vnaprej določenih ukazov menuja Okno. Zaradi podobne narave smo jih uredili v polje gradnikov, tako da za vse ukaze zadošča ena sama odzivna procedura:

```
' Različni načini urejanja, glede na menujsko izbiro
Private Sub mnuOknoUredi_Click(Index As Integer)
  Select Case Index
    Case 1
      ' Uredi okna vodoravno
      Arrange vbTileHorizontal
    Case 2
      ' Uredi okna navpično
      Arrange vbTileVertical
    Case 3
      ' Uredi okna kaskadno
      Arrange vbCascade
    Case 4
      ' Uredi ikone pomanjšanih oken
      Arrange vbArrangeIcons
  End Select
End Sub
```

Z urejanjem oken praktično nimamo nobenega dela. Vse postori pripravljena metoda **Arrange**, ki ji posredujemo enega od štirih parametrov.

Uporaba odložišča

Uporaba odložišča (*Clipboard*) je pri urejanju besedila pravzaprav nujna. Z njim lahko premikamo in prepisujemo kose besedila znotraj istega okna, med okni Urejevalnika in celo med različnimi okenskimi programi. V visual basicu je dostop do odložišča omogočen z vgrajenim predmetom **Clipboard**. Pozna samo naslednje metode:

Metoda	Opis
Clear	Zbriše vsebino odložišča.
GetData	Pobere morebitno grafično vsebino odložišča.
GetFormat	Preveri obliko vsebine odložišča.
GetText	Pobere morebitno besedilo z odložišča.
SetData	Postavi grafično vsebino na odložišče.
SetText	Postavi besedilo na odložišče.

S prenašanjem grafike na odložišče se bomo ukvarjali v naslednjem poglavju, tu pa si oglejmo, kako lahko uporabimo metode za delo z besedilom:

```
' Ukazi za delo z odložiščem
Private Sub mnuUrejanjeUkaz_Click(Index As Integer)
  With ActiveForm.txtBesedilo
    Select Case Index
      Case 1
        ' Izrez pomeni prenos na odložišče in brisanje
        Clipboard.SetText .SelText
        .SelText = ""
      Case 2
        ' Kopiranje pusti označeno besedilo pri miru
        Clipboard.SetText .SelText
      Case 3
        ' Prilepimo obstoječe besedilo na odložišču
        .SelText = Clipboard.GetText
    End Select
  End With
End Sub
```

Tudi vsi ukazi za delo z odložiščem so zbrani v polju gradnikov, zato jih obravnavamo znotraj ene same odzivne procedure. Vedno se sklicujemo na gradnik txtBesedilo znotraj dejavnega podrejenega obrazca. Vidimo, da je izbrano območje besedila označeno z lastnostjo **SelText**.

Pri uporabi vnosnih polj (*TextBox*) in kombiniranih zaprtih seznamov (*ComboBox*) v programski kodi imamo na voljo tri dodatne lastnosti. To so:

Lastnost	Tip	Pomen
SelStart	Celo število	Začetek označenega besedila.
SelLength	Celo število	Dolžina označenega besedila.
SelText	Niz	Vsebina označenega besedila.

Če želimo označiti vse besedilo v vnosnem polju, uporabimo dva programska stavka:

```
Text1.SelStart = 0
Text1.SelLength = Len(Text1.Text)
```

Tako **SelStart** kakor **SelLength** sta lahko najmanj 0. Prirejanje manjše vrednosti povzroči napako med izvajanjem. Z lastnostjo SelText lahko preberemo vsebino označenega dela, če pa ji priredimo novo vrednost, se označeni del *nadomesti* z novo vsebino, katere dolžina je poljubna.

Če zgled preizkusimo, lahko ugotovimo, da je mogoč prenos besedila znotraj okna, med različnimi podrejenimi okni, prav tako pa lahko prenašamo kose besedila med programi. Zmoti nas lahko podrobnost v menuju Urejanje. Vsi ukazi za delo z odložiščem so omogočeni, tudi če besedilo ni izbrano. Podobno je ves čas na voljo ukaz Prilepi, čeprav je morda odložišče prazno.

Težavo rešimo zelo učinkovito. Kakor vemo, *vsaki* menujski izbiri (razen ločilu) pripada odzivna procedura za dogodek Click. Torej je mogoče obdelati dogodek Click tudi za nadrejene menujske izbire, npr. za izbiro mnuUrejanje:

```
' O(ne)mogočim ustrezne podizbire
Private Sub mnuUrejanje_Click()
  ' Izreži in kopiraj sta omogočeni pri označenem besedilu
  mnuUrejanjeUkaz(1).Enabled = ActiveForm.txtBesedilo.SelLength > 0
  mnuUrejanjeUkaz(2).Enabled = mnuUrejanjeUkaz(1).Enabled
  ' Prilepi je omogočen le, če je na odložišču kakšno besedilo
  mnuUrejanjeUkaz(3).Enabled = Clipboard.GetFormat(vbCFText)
End Sub
```

Odzivna procedura se bo poklicala pred vsakim odpiranjem menuja Urejanje. Pokliče se celo, ko želi uporabnik sprožiti ukaz z bližnjico (npr. Ctrl+C). Zato je najprimernejše mesto za o(ne)mogočanje podrejenih menujskih izbir. Ukaza Izreži in Kopiraj sta na voljo le, če je v dejavnem podrejenem oknu izbrano besedilo (tedaj je lastnost SelLength zagotovo več od 0). Ukaz Prilepi omogočimo tedaj, ko je na odložišču besedilo. To ugotovimo s funkcijsko metodo **GetFormat** predmeta Clipboard.

Preiskovanje besedila

V menu Urejanje smo dodali tudi ukaza za preiskovanje besedila. Njuna izvedba je nadvse preprosta, saj je celotno besedilo samo dolg niz znakov. Za preiskovanje niza pa imamo na voljo funkcijo InStr, ki smo jo že spoznali pri Okenskem komandantu.

Prvi ukaz je Išči, ki začne preiskovati. Njegova odzivna procedura se glasi:

```
Private Sub mnuUrejanjeIsci_Click()
  ActiveForm.Iskanje
End Sub
```

Drugi ukaz, Išči znova, pa nadaljuje že začeto iskanje:

```
Private Sub mnuUrejanjeZnova_Click()
  ActiveForm.Iskanje False
End Sub
```

Kakor vidimo, je iskanje naloga podrejenega obrazca. Že iz klicev vidimo, da bo podprogram Iskanje uporabljal neobvezen parameter, saj ga prvič kličemo neposredno, drugič pa s parametrom. Zakaj je dobro iskanje prestaviti znotraj kode podrejenega obrazca?

Na splošno velja, da je bolje, če obrazec sam odgovarja za svojo vsebino. Če bi pozneje znotraj podrejenega obrazca prikazovali besedilo v drugačnem zapisu ali morda povsem novo vsebino, bi zadoščalo, da popravimo javno metodo iskanja, glavni obrazec pa bi bil brez sprememb uporaben še naprej.

> **NAMIG** V predmetnem žargonu pravimo, da ima podrejeno okno pripravljen *vmesnik*, prek katerega komunicira s preostalo kodo. Pri enakem vmesniku lahko različne vsebine nadziramo z isto programsko kodo. Vsebina podrejenega obrazca bi se lahko spremenila, npr. namesto besedila bi prikazovali preglednico števil. Če bi tudi v tem primeru razvili javno metodo Iskanje, bi lahko iz glavnega okna hkrati nadzirali podrejena okna z različnimi vsebinami. Preiskovanje bi bilo vedno izvedljivo prek enakega vmesnika – metode Iskanje. Tovrstni tehniki pravimo večličnost (*polymorphism*) in jo bomo izdatno uporabljali pri predmetih (*objects*).

Oglejmo si izpis javne metode podrejenega obrazca Iskanje:

```
' Iskanje po besedilu
Public Sub Iskanje(Optional blnVnos As Boolean = True)
Static strIskaniNiz
Dim intZacetek As Integer
Dim i As Long

  If blnVnos Or Len(strIskaniNiz) = 0 Then
    ' Iščemo od začetka
    strIskaniNiz = InputBox("Podaj iskani niz:", _
      "Iskanje", strIskaniNiz)
    intZacetek = 1
  Else
    ' Iščemo znova
    intZacetek = txtBesedilo.SelStart + 2
  End If
```

```
  If Len(strIskaniNiz) > 0 Then
    i = InStr(intZacetek, txtBesedilo.Text, strIskaniNiz, vbTextCompare)
    If i = 0 Then
      MsgBox "Iskanega niza" & vbCrLf & strIskaniNiz & _
        vbCrLf & "nisem našel!", vbInformation
    Else
      txtBesedilo.SelStart = i - 1
      txtBesedilo.SelLength = Len(strIskaniNiz)
    End If
  Else
    MsgBox "Podaj besedilo za iskanje!", vbCritical
  End If
End Sub
```

Neobvezni parameter določa, ali je treba od uporabnika zahtevati vnos iskane vsebine. Po privzeti nastavitvi ima resnično vrednost. Če se zahteva nov vnos ali če je niz z iskanim zaporedjem še nedoločen, vprašamo po iskani vsebini. Delo za nas opravi vgrajena funkcija InputBox. Iskanje je naloga funkcije InStr, ki išče od podanega začetnega mesta (to je 1 za prvi znak v nizu), tokrat pa smo prvič uporabili še četrti, neobvezni parameter, ki določa način primerjanja nizov. Konstanta vbTextCompare določa primerjanje znakov, brez vpliva njihove velikosti. Druga (in privzeta) možnost je vbBinaryCompare, ki je strožja, saj neposredno primerja kode znakov. Če je bilo iskanje uspešno, označimo najdeni del besedila. Lastnost SelStart mora kazati na znak pred najdenim, saj se njena vrednost začne z ničlo, medtem ko je prvi znak niza označen z zaporedno številko 1. Če bi izbrali ukaz Išči znova, bi začetek iskanja določili na podlagi trenutnega položaja v besedilu (lastnost SelStart). Ker ta lastnost kaže na mesto pred označenim besedilom, ga moramo povečati za 2, drugače bi znova našli isti niz.

Pravzaprav lahko pomen ukaza išči znova določimo na dva načina. Denimo, da imamo besedilo:

```
Peli smo: lalala, lala, lalala.
```

In iščemo podniz »lala«. Prvo iskanje najde in označi naslednji odlomek:

```
Peli smo: lalala, lala, lalala.
```

Kako naj se zdaj vede vnovično iskanje? Naj preskoči vse označeno besedilo:

```
Peli smo: lalala, lala, lalala.
```

Ali naj išče samo en znak naprej:

```
Peli smo: lalala, lala, lalala.
```

V zgornjem zgledu smo izbrali slednjo možnost. Če bi želeli iskati na prvi način, bi morali spremenljivko `intZacetek` nastaviti na naslednjo vrednost:

```
intZacetek = txtBesedilo.SelStart + txtBesedilo.SelLength
```

Prav zanimivo je spremljati, kakšne zmogljivosti so nam na voljo z navidez preprostim gradnikom, kot je vnosno polje. Pred nami pa je še nekaj nalog.

Delo z datotekami

Prvi zgled, program Okenski komandant, se je ukvarjal z upravljanjem datotek na višji ravni. V Urejevalniku bi radi z datotekami delali bolj neposredno. V njih bi radi pisali in iz njih brali obstoječo vsebino. Vsak program, ki želi uporabniku ponuditi delo z datotekami, mora priskrbeti mehanizem brskanja po datotečnem sistemu. Tokrat bomo uporabili najlažjo in verjetno tudi najzmogljivejšo pot: uporabili bomo zunanji gradnik za prikaz standardnih pogovornih oken, ki smo ga že spoznali.

Za začetek si oglejmo, kako se bomo odzvali na ustrezne menujske ukaze:

```
' Odprem datoteko
Private Sub mnuDatotekaOdpri_Click()
  OdpriBesedilo
  OmogociMenuje
End Sub

' Shranim besedilo
Private Sub mnuDatotekaShrani_Click()
  ShraniBesedilo
End Sub

' Shranim pod novim imenom
Private Sub mnuDatotekaShraniKot_Click()
  ShraniKot
End Sub
```

Delo smo znova preložili na lastne podprograme, kar je tudi najbolj smiselno. Najprej si natančneje oglejmo odpiranje datoteke.

Odpiranje

Za branje besedila bomo uporabili proceduro OdpriBesedilo:

```
' Procedura, ki prikaže standardno okno za odpiranje datotek
' in po potrebi pokliče proceduro za odpiranje novega okna
Private Sub OdpriBesedilo()
  With dlgCommonDialog
    ' Nastavimo parametre standardnega pogovora
    .Flags = cdlOFNFileMustExist + cdlOFNHideReadOnly
    .FileName = ""
    .DialogTitle = "Odpri"
    .CancelError = True
    ' In ga prikažemo
On Error GoTo Napaka
    .ShowOpen
    NovoOkno .FileName

Napaka:
  End With
End Sub
```

Procedura uporablja standardno pogovorno okno za odpiranje datotek. Najprej nastavi določen parameter, kot je npr. zahteva, da izbrana datoteka mora obstajati

(glej obravnavo gradnika zgoraj). Z nastavljeno lastnostjo CancelError bo prikazano pogovorno okno povzročilo ulovljivo napako, če uporabnik izbere gumb za preklic operacije. Zaenkrat predpostavimo potek brez napak, kajti natančnejši opis obravnavanja napak bomo še spoznali. Vidimo, da po uspešni izbiri datoteke njeno ime posredujemo proceduri NovoOkno. Kakor vemo, ta zaenkrat ne zna drugega, kakor da prikaže novo prazno okno. Očitno jo bomo morali posodobiti:

```
' Procedura, ki odpre novo okno, določeno s prototipom
' frmBesedilo in po potrebi pokliče funkcijo, ki iz datoteke
' prebere vsebino v vnosno polje
Private Sub NovoOkno(Optional strDatoteka)
Static intStevec As Integer
Dim frm As frmBesedilo

  ' Najprej povečamo števec okna, saj je prvič
  ' po privzeti vrednosti 0
  intStevec = intStevec + 1

  ' Spremenljivka kaže na predmet, ki se bo
  ' implicitno ustvaril ob prvem sklicevanju nanj

  Set frm = New frmBesedilo

  ' Nastavim posebne lastnosti obrazca
  ' glede na podano ime datoteke, ki je
  ' neobvezni parameter procedure
  If IsMissing(strDatoteka) Then
    ' Novo, prazno okno
    frm.strDatoteka = ""
    frm.blnNeimenovan = True
    frm.Caption = "Neimenovan" & intStevec
    frm.Show
  Else
    ' Podana datoteka, poizkusi jo prebrati
    frm.strDatoteka = strDatoteka
    frm.Caption = IzlusciDatoteko(strDatoteka)
    frm.blnNeimenovan = False
    If frm.PreberiDatoteko Then
      frm.Show
    Else
      Unload frm
    End If
  End If

End Sub
```

Zaradi združljivosti z že napisano kodo je edini parameter procedure neobvezen. Če ga ne podamo, se bo pač odprlo novo prazno okno. Vpeljali smo še statično spremenljivko, ki šteje novo odprta okna. To število uporabimo pri poimenovanju novih oken, tako da vsakemu pripade enolično določeno ime. Če proceduri pošljemo popolno ime datoteke, se vzpostavi novo okno. Njegov napis je ime datoteke brez poti (funkcijo IzlusciDatoteko smo razvili že pri Okenskem komandantu), njegovo vsebino pa določi nova funkcija, ki je javna metoda podrejenega obrazca. Če je njen izid uspešen, novo okno tudi prikažemo, drugače

pa se ga moramo znebiti. Zaradi samodejne stvaritve (ki jo določa rezervirana beseda New) se je obrazec ustvaril že pri prvem naslavljanju njegove lastnosti. Znebimo se ga s stavkom Unload.

Umazane podrobnosti so znova naloga podrejenega obrazca. Še enkrat ponovimo, da je opravila na nizki ravni najbolje prepustiti obrazcu. Če bi se kdaj v prihodnosti zamenjala oblika zapisa besedila, bi glavno okno lahko uporabili brez sprememb, posodobili bi samó funkcijo PreberiDatoteko. Ta se zaenkrat glasi (funkcija je javni del obrazca frmBesedilo):

```
' Javna funkcija, ki prebere besedilo iz datoteke v vnosno polje.
' Vrnjena vrednost predstavlja uspešnost operacije.
Public Function PreberiDatoteko() As Boolean
Dim intFile As Integer

On Error GoTo Napaka
  intFile = FreeFile
  ' Odpremo izbrano datoteko za branje
  Open strDatoteka For Input As #intFile
  ' Prikažemo kazalec čakanja
  Screen.MousePointer = vbHourglass
  ' Preberemo vsebino datoteke neposredno v lastnost
  txtBesedilo.Text = Input(LOF(intFile), #intFile)
  ' Sprememba povzroči dogodek Change, zato ponastavimo
  blnSprememba = False
  Close #intFile
  ' Običajni kazalec miške
  Screen.MousePointer = vbDefault
  PreberiDatoteko = True
  Exit Function

Napaka:
  ' Pospravimo
  Close #intFile
  Screen.MousePointer = vbDefault
  MsgBox "Napaka pri odpiranju!" & vbCrLf & Err.Description, vbCritical
  PreberiDatoteko = False
End Function
```

Visual basic lahko obravnava datoteke na tri načine, kot:

- binarne datoteke (*binary files*),
- zaporedne datoteke (*sequential files*) in
- datoteke z naključnim dostopom (*random access files*).

Prvi je najzmogljivejši, vendar tudi programersko najzahtevnejši. Uporabili ga bomo v naslednjem poglavju. Drugi način uporabimo za datoteke z besedilom, torej je kakor nalašč za nas. Tretji način je primeren za zapis uporabniško definiranih struktur. Odkar zna visual basic neposredno uporabljati zbirke podatkov, je taka raba navadnih datotek postala praktično odveč.

Vsako datoteko moramo pred uporabo odpreti v želenem načinu. Pri odpiranju ji priredimo celo število, ki v vsej nadaljnji kodi predstavlja odprto datoteko. To število imenujemo tudi *datotečna ročica* (*file handle*). Čeprav bi lahko izbrali poljubno celo število, je koda mnogo bolj trdoživa, če ustrezno število določimo z vgrajeno funkcijo **FreeFile**, ki vedno vrne še neuporabljeno ročico.

Datoteko odpremo s stavkom **Open**, katerega popolna oblika za *zaporedni* način dostopa je:

```
Open pot For {Input|Output|Append} _
[{Shared|Lock[ {Read|Write|Read Write}]] As [#]ročica
```

Zaporedno datoteko lahko odpremo za branje (Input), pisanje (Output) ali za dodajanje nove vsebine na konec datoteke (Append). Po želji lahko določimo dostopnost do iste datoteke za druge procese (*sharing*). Dostop lahko drugim povsem prepovemo (Lock Read Write), dopustimo zgolj branje (Lock Write) ali pisanje (Lock Read) ali pa datoteko odpremo v deljenem načinu (ključno besedo izpustimo ali pa zapišemo Shared). Na koncu predpišemo še ročico. Starejše različice basica so zahtevale pred ročico predpono #. Zaradi lažje berljivosti kode jo uporabljamo tudi v zgornjem zgledu, čeprav v visual basicu ni več nujna.

Datoteko smo torej odprli za branje v deljenem načinu in ji priredili ročico, shranjeno v spremenljivki `intFile`. Ker je branje iz datoteke lahko daljši proces, začasno spremenimo kazalec miške v peščeno uro. Pri tem nam pomaga lastnost vgrajenega predmeta Screen, ki ga bomo podrobneje spoznali v naslednjem poglavju.

Branje iz zaporedne datoteke je mogoče z eno funkcijo in dvema stavkoma, katerih opis je v spodnji tabeli:

Funkcija/Stavek	Uporaba	Opis
Funkcija **Input**	`Input(številoZnakov, ročica)`	Vrne niz s prebranim številom znakov.
Stavek **Input #**	`Input #ročica, spremenljivka [, spremenljivka…]`	Prebere vrednosti poljubnega števila spremenljivk. Te vrednosti bi moral zapisati stavek **Write #**.
Stavek **Line Input #**	`Line Input #ročica`	Prebere vrstico besedila[33].

Najuporabnejša je prav funkcija Input, ki smo jo uporabili tudi zgoraj in v enem stavku prebrali vsebino vse datoteke. Na pomoč smo poklicali funkcijo **LOF** (*Length Of File*), ki vrne celotno dolžino odprte datoteke v znakih.

[33] Konec vrstice predstavlja znak CR (*carriage return*, v abecedi ASCII mu pripada koda 13) ali kombinacija CRLF (*carriage return-linefeed* z ASCII kodama 13 in 10).

Po uspešnem branju obnovimo kazalec miške. Vsako odprto datoteko *moramo* na koncu tudi ročno zapreti. To postori stavek Close. Tudi pri branju smo vključili obravnavanje napak. Z natančnejšo razlago pa bomo še čisto malo počakali.

Zapisovanje

Pri shranjevanju datotek smo uporabili dve funkciji: ShraniBesedilo in ShraniKot. Prva je javna, ker nam bo prišla prav tudi v podrejenem obrazcu, druga pa je zasebna. Oglejmo si njun izpis (funkciji sta del glavnega obrazca):

```
' Funkcija, ki poizkusi shraniti besedilo pod znanim imenom.
' Če ime ni znano, pokliče funkcijo ShraniKot()
Public Function ShraniBesedilo() As Boolean
  With ActiveForm
    If .blnNeimenovan Then
      ' Nepoimenovane datoteke ne moremo
      ' preprosto shraniti, potrebujemo ime
      ShraniBesedilo = ShraniKot
    Else
      ' Poimenovano datoteko pa lahko
      ShraniBesedilo = .ZapisiDatoteko
    End If
  End With
End Function

' Funkcija, ki prikaže standardno pogovorno okno za
' shranjevanje datotek in po potrebi pokliče funkcijo
' za zapis besedila vnosnega polja v datoteko
Private Function ShraniKot() As Boolean
  With dlgCommonDialog
    ' Nastavimo parametre standardnega pogovora
    .Flags = cdlOFNOverwritePrompt + _
      cdlOFNPathMustExist + cdlOFNHideReadOnly
    .FileName = ActiveForm.strDatoteka
    .DialogTitle = "Shrani kot"
    .CancelError = True
    ' In ga prikažemo
On Error GoTo Napaka
    .ShowSave
    ' Datoteko lahko zapišemo pod novo izbranim imenom
    With ActiveForm
      .strDatoteka = frmGlavni.dlgCommonDialog.FileName
      .Caption = frmGlavni.dlgCommonDialog.FileTitle
      .blnNeimenovan = False
      ' Vrnemo izid shranjevanja
      ShraniKot = .ZapisiDatoteko
    End With
    Exit Function

Napaka:
    ' Sporočimo preklicano shranjevanje
    ShraniKot = False
  End With
End Function
```

Funkcija ShraniBesedilo ni nič posebnega. Če je ime datoteke že znano, pokliče javno metodo podrejenega obrazca, zadolženo za shranjevanje. Če ime datoteke še ni bilo določeno, preloži delo funkciji ShraniKot, ki vedno zahteva ime datoteke. Pri tem ji pomaga standardno pogovorno okno za shranjevanje. Tokrat smo nastavili nekaj novih parametrov. Predvsem je pomembno določilo, ki uporabnika opozori, kadar želi prepisati že obstoječo datoteko. Zahtevamo tudi, da izbrana pot (torej mapa) že obstaja. Če uporabnik shranjevanja ne prekliče, popravimo ime datoteke in ustrezen napis na oknu. Na koncu pokličemo javno metodo podrejenega obrazca za shranjevanje.

Funkcija ZapisiDatoteko je določena kot javna metoda obrazca frmBesedilo. Njen izpis je:

```
' Javna funkcija, ki poskrbi za zapis besedila vnosnega polja
' v datoteko. Vrnjena vrednost predstavlja uspešnost operacije.
Public Function ZapisiDatoteko() As Boolean
Dim intFile
Dim strBesedilo As String

On Error GoTo Napaka
  intFile = FreeFile

  ' Odpremo datoteko za pisanje
  Open strDatoteka For Output Lock Read Write As #intFile
  ' Besedilo shranimo v spremenljivko
  strBesedilo = txtBesedilo.Text
  ' Prikažemo kazalec čakanja
  Screen.MousePointer = vbHourglass
  ' Shranimo vsebino spremenljivke v datoteko
  Print #intFile, strBesedilo
  Close #intFile
  ' Običajni kazalec miške
  Screen.MousePointer = vbDefault
  blnSprememba = False
  ZapisiDatoteko = True

  Exit Function

Napaka:
  ' Pospravimo
  Close #intFile
  Screen.MousePointer = vbDefault
  MsgBox "Napaka pri shranjevanju!" & vbCrLf & _
    Err.Description, vbCritical
  ZapisiDatoteko = False

End Function
```

Delo je zelo podobno branju, le da smo sedaj zaporedno datoteko odprli za pisanje in preprečili hkraten dostop za druge procese. Če je zaporedna datoteka odprta za pisanje, bo novi zapis popolnoma prekril prej obstoječo vsebino, ki je s tem izgubljena. Za dodajanje nove vsebine na konec datoteke imamo na voljo posebno odpiranje (Append).

Za zapis v zaporedno datoteko imamo na voljo dva stavka:

Stavek	Uporaba	Opis
Stavek **Write #**	`Write #ročica, spremenljivka [, spremenljivka…]`	Zapiše poljubno zaporedje spremenljivk. Tovrsten zapis pravilno prebere stavek Input #.
Stavek **Print #**	`Print #ročica, niz`	Zapiše celoten niz v zaporedno datoteko.

NAMIG Stavka Write # in Input # sta ohranjena iz dediščine Microsoftove različice programskega jezika basic. Namenjena sta preprostemu zapisovanju najrazličnejših podatkovnih tipov v datoteko z besedilom (*plain text file*). Naslednji preprost zgled:

```
intA = 125
sngA = 3.42
blnA = False
strA = "Besedilo"
vntA = Now

Open "c:\test.txt" For Output As #1
Write #1, intA, sngA, blnA, strA, vntA
Close #1
```

bi v datoteko c:\test.txt zapisal naslednje »besedilo«:

```
125,3.42,#FALSE#,"Besedilo",#1998-05-06 15:50:37#
```

Tovrstni zapis najhitreje preberemo z ustreznim stavkom Input #.

```
Open "c:\test.txt" For Input As #1
Input #1, intA, sngA, blnA, strA, vntA
Close #1
```

Njuna uporaba je zanimiva predvsem za shranjevanje različnih nastavitev, ki jih ne želimo zapisovati v register.

Sledenje spremembam

Pri branju in shranjevanju v datoteko smo se ukvarjali tudi z zasebno spremenljivko `blnSprememba`, ki ponazarja potrebo po shranjevanju sprememb. Lepo vzgojen program mora spremljati uporabnika in mu ne sme dopustiti, da bi po nerodnosti izgubil vnesene podatke.

Omenjena spremenljivka je ob vzpostavitvi obrazca nastavljena na neresnično vrednost. Taka ostane tudi ob odpiranju novega okna, kadar datoteko odpremo ali shranimo, pa jo celo eksplicitno znova postavimo na neresnično vrednost. Kdaj pravzaprav označimo, da je v besedilu prišlo do sprememb? Prav preprosto, za to nam rabi dogodek Change, ki se povzroči ob vsaki spremembi vsebine vnosnega polja. V podrejeni obrazec torej dodamo naslednjo odzivno proceduro:

```
' Sprememba besedila
Private Sub txtBesedilo_Change()
  ' Označimo spremembo
  blnSprememba = True
End Sub
```

Sedaj nam ne bo ušla nobena sprememba. Kje pa je pravo mesto, da uporabnika opozorimo na shranjevanje spremenjenega besedila? Tudi ta dogodek že poznamo, imenuje se Query_Unload:

```
' Zadnja možnost za preklic zapiranja okna
Private Sub Form_QueryUnload(Cancel As Integer, UnloadMode As Integer)
Dim intIzid As Integer

  ' Če je prišlo do spremembe
  If blnSprememba Then
    intIzid = MsgBox("Besedilo dokumenta " & Caption & _
      " se je spremenilo." & vbCrLf & _
      "Shranim spremembe?", vbYesNoCancel + vbQuestion, _
      "Opozorilo")
    If intIzid = vbYes Then
      ' Uporabnik je želel shraniti dokument, vendar
      ' shranjevanje ni uspelo. Prekličemo zapiranje.
      If Not frmGlavni.ShraniBesedilo Then Cancel = True
    ElseIf intIzid = vbCancel Then
      ' Uporabnik je takoj preklical zapiranje
      Cancel = True
    End If
  End If
End Sub
```

Če je uporabnik v pogovornem oknu izbral preklic ali če je shranjevanje spodletelo, prekličemo zapiranje okna. Ta preklic ima lahko daljnosežne posledice:

- Če je uporabnik želel zapreti podrejeno okno, se to ne bo zaprlo.
- Če je uporabnik želel zapreti ves program in samo v enem podrejenem oknu prekličemo zapiranje, se tudi ves program ne bo končal.
- Če je uporabnik izbral vnovični zagon operacijskega sistema in prekličemo zapiranje samo enega podrejenega okna v našem programu, se tudi ves operacijski sistem ne bo znova zagnal[34].

Pri načrtovanju vmesnika smo v podrejeni obrazec uvrstili tudi oznako (*Label*). Njej pripada vloga vrstice stanja. Zaenkrat jo bomo uporabili, da nam nakazuje vsebino spremenljivke `blnSprememba`, torej nam bo nakazovala morebitni popravek besedila. Besedilo oznake bo določala nova, zasebna procedura ObnoviStanje:

```
Private Sub ObnoviStanje()
  lblStatus.Caption = IIf(blnSprememba, "(Spremenjeno)", "")
End Sub
```

[34] V resnici ponudijo Okna vsem uporabniškim programom nek čas, v katerem se morajo spodobno zaključiti ali pa preklicati zapiranje. Če tega v doglednem času ne storijo, jih lahko prisilno zaključi.

Proceduro moramo klicati z vseh mest, ki lahko spremenijo spremenljivko `blnSprememba`, prav gotovo pa jo moramo poklicati tudi ob prikazu obrazca, v odzivu na dogodek Activate:

```
' Obnovim stanje ob prikazu obrazca
Private Sub Form_Activate()
  ObnoviStanje
End Sub
```

Dopolniti moramo še odziv na dogodek Change:

```
' Sprememba besedila
Private Sub txtBesedilo_Change()
  ' Označimo spremembo
  blnSprememba = True
  ObnoviStanje
End Sub
```

In tudi funkcijo shranjevanja, ki spremembe shrani in s tem izniči:

```
Public Function ZapisiDatoteko() As Boolean
' ...
  blnSprememba = False
  ObnoviStanje
  ZapisiDatoteko = True
' ...
End Function
```

Prestrezanje napak

Pri prikazu standardnih pogovornih oken in znotraj kritičnih operacij, kot je shranjevanje in odpiranje datotek, smo že lahko opazili kodo, ki omogoča prestrezanje izvajalnih napak (*run time errors*). Če prestrezanja napak ne omogočimo, je vsaka izvajalna napaka *usodna*. Usodna napaka (*fatal error*) pomeni konec izvajanja kode, podoben vsiljeni prekinitvi. Če program preizkušamo v razvojnem okolju, je mogoče usodno napako včasih odpraviti z ročnim posredovanjem. Dokončno preveden samostojni program ob usodni napaki nepreklicno konča izvajanje.

V visual basicu ni mogoče napisati osrednjega podprograma, v katerem bi se obravnavale vse izvajalne napake. Napake moramo vedno obravnavati na ravni podprograma, ki mu lahko dodamo *prestrezovalnik* izvajalnih napak (*run time error handler*). Prestrezovalnik izvajalnih napak omogočimo s stavkom

`On Error Goto` *oznaka*

Oznaka je simbolična označitev neke programske vrstice znotraj istega podprograma. Oznaka (*label*) v visual basicu je pravilno simbolično ime, ki mu sledi dvopičje (`:`). Prestrezovalnik napake lahko naknadno izključimo s stavkom, v katerem oznako nadomesti ničla:

```
On Error GoTo 0
```

Oglejmo si zgled:

```
Sub KriticenPodprogram

On Error GoTo Napaka
  ' Kritični del podprograma
On Error GoTo 0
  ' Nekritični del

  ' Predčasno končamo izvajanje kode,
  ' sicer bomo vedno padli v prestrezovalnik napake
  Exit Sub

Napaka:
  ' Prestrezovalnik izvajalne napake

End Sub
```

Če v kritičnem delu podprograma pride do izvajalne napake, visual basic nemudoma pokliče prvi stavek v prestrezovalniku. Prestrezovalnik lahko poizkusi napako odpraviti, potem pa ima na voljo več možnosti:

- Lahko poizkusi ponoviti stavek, ki je povzročil napako. Ponovitev kritičnega stavka dosežemo s stavkom:

  ```
  Resume
  ```

- Lahko nadaljuje izvajanje na stavku, ki sledi. Vrnitev nanj dosežemo s stavkom

  ```
  Resume Next
  ```

- Lahko prenese izvajanje na vrstico, označeno z novo oznako. Preskok dosežemo s stavkom

  Resume *oznaka*

> **NAMIG** Pri uporabi stavka **Resume** moramo biti zelo previdni, saj lahko hitro povzroči neskončno zanko. Preden ga pokličemo, mora obstajati možnost, da je napaka odpravljena. Značilen zgled je dostop do disketnega pogona. Če ta ni pripravljen, se sproži izvajalna napaka. V prestrezovalniku obvestimo uporabnika, da mora vstaviti disketo v pogon. To stori funkcija MsgBox, ki zaradi svoje modalnosti začasno prekine izvajanje. Problematičen stavek poizkusimo ponoviti samo, ko uporabnik opozorilo potrdi.

Če v prestrezovalniku ne napišemo drugega kakor stavek **Resume Next**, ki nadaljuje izvajanje na naslednjem stavku, lahko ves prestrezovalnik napake zapišemo še preprosteje:

```
On Error Resume Next
```

Tako zapisani prestrezovalnik pravzaprav to niti ni. Če pride do napake, se izvajanje preprosto preseli na naslednji programski stavek. Koda je zato nadvse trdoživa, saj je nobena izvajalna napaka ne vrže iz tira. Smiselnost takega početja pa je pogosto vprašljiva, saj se nam lahko izmuznejo nevarne logične napake, ki bi jih bilo dobro prestreči.

V določenem primeru pa je raba stavka On Error Resume Next priporočljiva. To je pri uporabi zunanjih predmetov, kot so npr. zunanji gradniki ActiveX:

```
' Vzklopimo prestrezovalnik
On Error Resume Next
' Kritični del podprograma, ki uporablja zunanji predmet

' Izklopimo prestrezovalnik
On Error Goto 0

' Preverimo vzrok napake
If Err.Number Then
  If Err.Source = App.Title
  ' Napaka je v našem projektu
  Else
  ' Vzrok napake je zunaj
  End If
End If
```

Trdoživi prestrezovalnik napake smo vklopili le za kratek čas. Morebitno napako analiziramo pozneje, z vgrajenim predmetom **Err**.

Mar moramo zaradi obravnavanja izvajalnih napak znotraj podprogramov za popolnoma trdoživ program zapisati prestrezovalnik v prav vseh podprogramih? Na srečo ne, saj morebitna izvajalna napaka potuje nazaj po *klicnem skladu*, tako da zadošča prestrezovalnik v vseh *odzivnih* podprogramih, ki so vedno na vrhu tega sklada. Obravnavo klicnega sklada si bomo ogledali pri metodi Raise.

Predmet Err

Vgrajeni predmet Err vsebuje lastnosti, ki izčrpno opišejo napako, pozna pa tudi dve metodi, ki nam pomagata pri ročnem *proženju* napak (*raising errors*). V naslednjih dveh tabelah so naštete lastnosti in metodi predmeta Err:

Lastnost	Tip	Opis
Number	Celo število	Številčna oznaka napake.
Source	Niz	Izvor napake.
Description	Niz	Opis napake.
HelpFile	Niz	Popolna pot do datoteke s pomočjo.
HelpContext	Niz	Oznaka odgovarjajoče teme pomoči.
LastDLLError	celo število	Oznaka zadnje napake pri klicu deljene knjižnice (DLL).

Metoda	Opis
Clear	Zbriše informacijo o zadnji napaki.
Raise	Sproži ročno ustvarjeno napako.

Oglejmo si zgled prestrezovalnika, ki nas izčrpno obvesti o napaki in ponudi morebitno pomoč:

```
With Err
  MsgBox .Source & " je povzročil napako " & _
    .Description & " s kodo " & .Number, , _
    "Napaka!", _
    .Helpfile, _
    .HelpContext
  .Clear
End With
```

Predmet Err je vgrajen v visual basic in je na voljo v samo enem izvodu, zato vedno hrani le informacijo z opisom zadnje izvajalne napake. Po želji lahko to informacijo ročno zbrišemo z metodo **Clear**. Samodejno jo briše vsaka oblika stavkov Resume in Exit.

Proženje napak

Včasih nam pride prav možnost ročnega proženja izvajalne napake z metodo **Raise**. Denimo, da iz podprograma A kličemo podprogram B. Podprogram B ima definiran prestrezovalnik napake in pokliče podprogram C. Če bi radi iz podprograma C opozorili podprogram B, da nekaj ni v redu, lahko ročno sprožimo izvajalno napako. Ta bo potovala nazaj po *klicnem skladu*, dokler ne naleti na prestrezovalnik, v našem primeru nazaj do podprograma B. Če se v podprogramu B *znotraj* prestrezovalnika napake povzroči nova izvajalna napaka, bo ta potovala nazaj v podprogram A.

Slika 9-13: Klicni sklad Urejevalnika v trenutku, ko vstopimo v proceduro odpiranja datoteke.

NAMIG V klicnem skladu je zaporedje podprogramov, ki so klicali drug drugega. V zgoraj opisanem zgledu bi pri vstopu v podprogram C klicni sklad vseboval podprograma B in A. S klicnim skladom visual basic ve, da se mora po izteku podprograma C najprej vrniti v B in šele zatem v A. Do pomika po skladu nazaj pride, kakor smo ravnokar

opisali, tudi pri izjemah. Klicni sklad lahko pregledujemo samo med prekinitvijo izvajanja programa. Prikaže ga ukaz **View ➪ Call Stack**.

Mehanizem ročnega proženja izjem kot način sporočanja nepravilnih stanj programa je možnost, ki precej izboljša berljivost programske kode. Oglejmo si naslednji hipotetični zgled:

```
Function PrepisiDatoteko(a,b)
  PrepisiDatoteko = False
  If PravilnoIme(a) Then
    If PravilnoIme(b) Then
      If DovoljProstora(b) Then
        If OdpriDatoteko(a) Then
          If Prepis(a,b) Then PrepisiDatoteko = True
        Else
          MsgBox "Napaka pri odpiranju"
        End If
      Else
        MsgBox "Premalo prostora"
      End If
    Else
      MsgBox "Nepravilno ime " & b
    End If
  Else
    MsgBox "Nepravilno ime " & a
  End If
End Function
```

Zgornja funkcija je zapisana »klasično«, z veliko preverjanja v gnezdenih stavkih If, saj vsaka napaka onemogoči uspešno prepisovanje datoteke.

Drug pristop je mogoč s proženjem lastnih napak:

```
Function PrepisiDatoteko(a,b)
On Error Goto napaka
  If Not PravilnoIme(a) Then _
    Err.Raise vbObjectError + 1, , "Nepravilno ime"
  If Not PravilnoIme(b) Then _
    Err.Raise vbObjectError + 1, , "Nepravilno ime"
  If Not DovoljProstora(b) Then _
    Err.Raise vbObjectError + 2, , "Premalo prostora"
  If Not OdpriDatoteko(a) Then _
    Err.Raise vbObjectError + 3, , "Ne morem odpreti datoteke"
  PrepisiDatoteko = Prepis(a,b)
Exit Function

napaka:
  MsgBox "Napaka " & Err.Description
  PrepisiDatoteko = False

End Function
```

Gnezdeni stavki If sedaj niso več potrebni, saj sprožena napaka nemudoma preseli izvajanje v prestrezovalnik in do prepisovanja sploh ne pridemo.

Zgled bi bil lahko še preprostejši, če bi vse uporabljene funkcije še same prožile napake, ko bi prišlo do nepravilnega stanja (seveda bi ne smele imeti lastnega prestrezovalnika, da bi napaka po klicnem skladu »izplavala ven« v naši funkciji):

```
Function PrepisiDatoteko(a,b)
On Error Goto napaka
  PravilnoIme a
  PravilnoIme b
  DovoljProstora b
  OdpriDatoteko a
  PrepisiDatoteko = Prepis(a,b)
Exit Function

napaka:
  MsgBox "Napaka " & Err.Description
  PrepisiDatoteko = False

End Function
```

Metoda Raise zahteva en parameter: kodo povzročene napake. Po želji lahko naštejemo še več parametrov, ki zapolnijo vsako lastnost predmeta Err (glej zgoraj). Kadar želimo uporabiti lastno oznako napake, uporabimo kodo, ki se začne na vgrajeni konstanti vbObjectError. S tem se izognemo uporabi katere od rezerviranih kod, ki opisujejo običajne izvajalne napake v visual basicu. Vse te kode so naštete v sprotni pomoči. Praktični zgled proženja lastne napake bom spoznali v naslednjem poglavju.

NAMIG Programsko kodo, razvito do te točke, najdete na priloženi plošči CD-ROM v mapi \Primeri\2-Delavnica\2-Urejevalnik\3-Koda\.

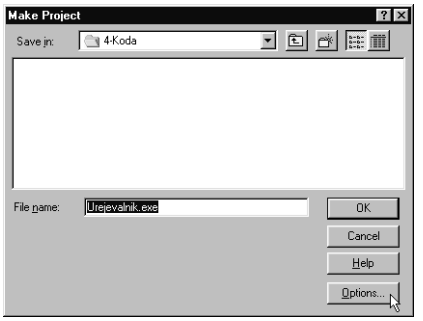
Prevajanje in podpis

Razvili smo precej uporaben program, ki je resen konkurent okenski Beležki. Čas je, da jo prevedemo v samostojno obliko, ki se izvaja precej hitreje. Po preizkusu izvajanja zunaj razvojnega okolja jo bomo opremili še z *vizitko*.

Prevajanje

Vsak zaokrožen projekt v visual basicu vrste standard EXE lahko prevedemo v samostojen, izvršljiv program. Operacija je zelo preprosta. V menuju **File** izberemo ukaz **Make Urejevalnik.exe...** (ime ukaza je odvisno od imena dejavnega projekta). Do samostojnega programa najhitreje pridemo, če v prikazanem pogovornem oknu izberemo gumb OK.

Slika 9-14: Pogovorno okno ukaza Make exe

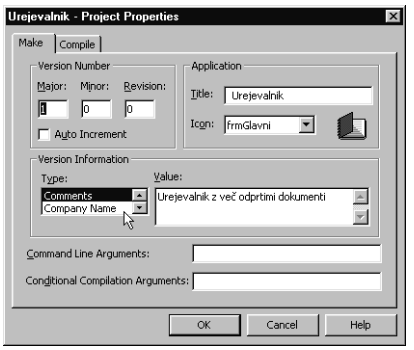


Po želji lahko izberemo gumb **Options...** in pred prevajanjem nastavimo še nekaj »malenkosti«. Dodatni parametri, ki opisujejo projekt, so zbrani v novem pogovornem oknu z dvema jezičkoma. Prva stran (jeziček Make) ponuja naslednje nastavitve:

- Oznaka številke različice (okvir **Version Number**). Po želji lahko vklopimo stikalo **Auto Increment** in visual basic bo samodejno povečal številko revizije ob vnovičnem prevajanju.
- Naziv in ikono programa (okvir **Application**). Tu določimo opisno ime programa, pod katerim bo predstavljena operacijskemu sistemu, in ikono programa, ki je lahko ikona poljubnega obrazca v projektu.
- Dodatne informacije o projektu (okvir **Version Information**). S seznama Type lahko izberemo več opisnih parametrov projekta. Njihovo vrednost določimo v vnosnem polju Value. Vse te vrednosti se zapišejo v prevedeno datoteko. Lahko jih pregledujemo npr. v lastnostih datoteke, ki jih prikaže Raziskovalec (*Explorer*).

Za naš projekt bomo določili parametra Comments in Legal Copyright. Prvi naj z več besedami opiše namen programa, npr. »Urejevalnik z več odprtimi dokumenti«. Drugi parameter opisuje lastnika avtorskih pravic programske kode.

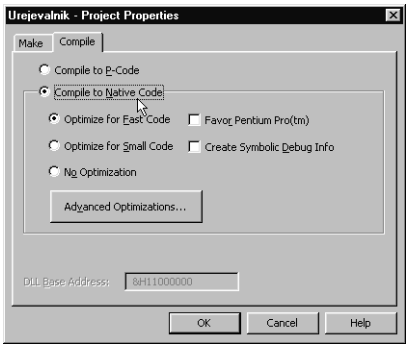
Slika 9-15: Osnovne nastavitve projekta



Druga stran pogovornega okna (jeziček Compile) omogoča nastavitev posebnih možnosti pri prevajanju kode. Najpomembnejša je izbira prevajanja v vmesno (P-Code) ali domorodno kodo (Native Code). Programi, prevedeni v domorodno kodo, se izvajajo bistveno hitreje. Na voljo imamo še izbiro optimalizacije za hitrost (Speed) ali velikost (Size). Po želji lahko določimo še nekaj zahtevnejših optimalizacijskih postopkov – skrivajo se pod gumbom **Advanced Optimizations**, vendar ti lahko povzročijo nepravilno izvajanje kode. Na voljo sta še dve stikali:

- Stikalo **Favor Pentuim Pro** izbere posebno optimalizacijo za procesor Pentium Pro. Tako prevedeni programi bodo pri drugih procesorjih delovali malo počasneje, na Pentiumu Pro pa nekoliko hitreje.
- Vklopljeno stikalo **Create Symbolic Debug Info** naroči prevajalniku, naj v prevedeno datoteko vključi napotke za zunanje razhroščevanje znotraj razvojnega okolja Microsoft Developer Studio.

Slika 9-16: Določanje možnosti pri prevajanju kode



Po prevajanju dobimo na disku novo izvršljivo datoteko .EXE, ki jo lahko poženemo tudi zunaj izvajalnega okolja. Izvaja se nekoliko hitreje, dejanski pospeški so predvsem odvisni od namena programa. Računanje s celimi števili je skoraj dvajsetkrat (20×) hitrejše, dostop do datotek ali zbirk podatkov pa se bistveno ne pospeši, saj je odvisen od drugih dejavnikov.

> **NAMIG** Nastavitve parametrov projekta lahko v celoti opravimo tudi brez prevajanja. Z izbiro zadnjega ukaza v menuju **Project** (**Urejevalnik properties...**, ime je odvisno od dejavnega projekta), se prikaže pogovorno okno s kar štirimi stranmi. Vmesni dve sta enaki zgoraj opisanima, na voljo pa imamo tudi nastavitev vrste projekta, povezane datoteke pomoči in še nekatere zahtevnejše zmožnosti.

Podpis

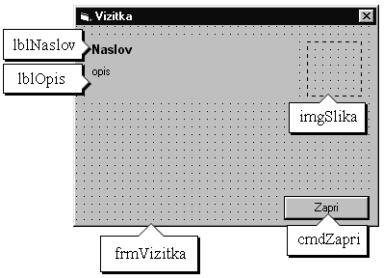
Programe pogosto opremljamo z *vizitko*. Vizitka v okenskem žargonu pomeni posebno pogovorno okno (*about window*), ki ponudi osnovne podatke o programu.

V nadaljevanju bomo razvili predlogo vizitke, ki jo praktično brez popravkov lahko vključimo v poljuben projekt.

> **NAMIG** Če se želite spopasti s precej zapleteno vizitko, imate med pripravljenimi predlogami eno že na voljo. Z izbiro ukaza **Project ➪ Add Form** prikažete galerijo pripravljenih obrazcev, v kateri izberete predlogo **About Dialog**.

Uporabniški vmesnik bodoče vizitke je prikazan na spodnji sliki:

Slika 9-17: Vmesnik vizitke



Lastnosti gradnikov, ki ga sestavljajo, so naštete v spodnji tabeli:

Gradnik	Lastnost	Nastavitev
obrazec (*Form*)	(Name)	frmVizitka
	BorderStyle	3 - Fixed Dialog
	Caption	"Vizitka"
	StartUpPosition	2 - CenterScreen
gumb (*CommandButton*)	(Name)	cmdZapri
	Cancel	True
	Caption	"Zapri"
	Default	True
oznaka (*Label*)	(Name)	lblNaslov
	Caption	"Naslov"
	Font	(Arial, 10, Bold)
oznaka (*Label*)	(Name)	lblOpis
	Caption	"opis"
slika (*Image*)	(Name)	imgSlika

Na vmesniku smo uporabili nov gradnik, sliko (*Image*). Gradnik ima dve funkciji:

- Vanj lahko shranimo sliko, ki jo pozneje uporabimo iz programske kode. V tem primeru je gradnik ponavadi neviden.
- Okrasna funkcija. Gradnik popestri videz uporabniškega vmesnika.

Visual basic ima na voljo precej uporabnejši zunanji gradnik, ki omogoča shranjevanje in obdelovanje večjega števila slik, zato je prva funkcija gradnika Image čedalje manj pomembna. Tudi v našem zgledu bo gradnik rabil zgolj v okrasne namene.

Sliko v gradniku lahko določimo med načrtovanjem, z lastnostjo **Picture**. Druga možnost je branje slike iz zunanje datoteke (tu nam pomaga funkcija **LoadPicture**). Tretja možnost je branje slike iz datoteke sredstev. Branje slike bomo spoznali v naslednjem poglavju.

Napišimo še dve odzivni proceduri. Obdelali bomo dogodek Load in klik ukaznega gumba:

```
' Začetne nastavitve
Private Sub Form_Load()
  Icon = Forms(0).Icon
  With App
    Caption = .Title
    lblNaslov.Caption = .Title & " v" & .Major & "." & .Minor & ""
    lblOpis.Caption = App.Comments & vbCrLf & App.LegalCopyright
  End With
  imgSlika.Picture = Icon
End Sub

' Privzeti in preklicni gumb skrije okno
Private Sub cmdZapri_Click()
  Unload Me
End Sub
```

Podprograma sta napisana v kodi, ki se prilagodi vsakemu projektu. Torej kakor nalašč za predlogo. V dogodku Load obrazcu določimo ikono, ki jo prevzamemo iz prvega obrazca. Vsi obrazci programa so vedno navedeni v zbirki Forms. Za določitev besedila oznak posežemo po vgrajenem predmetu **App**. Ta pozna kup lastnosti, ki natančno opišejo program, ter dve metodi, ki omogočata shranjevanje *dnevnika* (*log*). Natančneje ga bomo spoznali v nadaljevanju. Gradnik slika pozna lastnost Picture, ki ji lahko priredimo poljubno bitno sliko. Mi uporabimo kar ikono obrazca, ki smo jo že prej uskladili z ikono glavnega okna. Klik gumba obdelamo tako, da obrazec odstranimo. S tem karseda zmanjšamo pomnilniške zahteve vizitke.

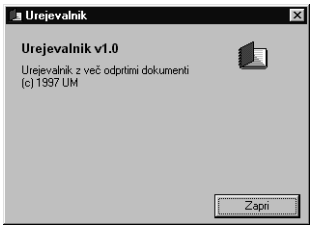
Tako pripravljeno vizitko lahko shranimo kot predlogo v mapo Forms (natančnejše napotke za pripravo predloge smo spoznali pri pripravi glavnega obrazca vmesnika MDI). Če jo bomo shranili kot predlogo, jo še zapremo in odpremo nov obrazec, temelječ na tej predlogi.

Oglejmo si, kako lahko pripravljen obrazec uporabimo v Urejevalniku. Že prej smo v menuju Pomoč določili ukaz Vizitka. Sedaj ga opremimo še z ustrezno odzivno proceduro:

```
Private Sub mnuPomocNasveti_Click()
  frmVizitka.Show vbModal, Me
End Sub
```

Preizkus novega okna je razviden s spodnje slike. Izid je odvisen od nastavljenih parametrov projekta (glej zgoraj, Prevajanje), ki se upoštevajo v besedilu oznak.

Slika 9-18: Vizitka Urejevalnika



Predmet App

Predmet App je vgrajeni predmet visual basica, ki s svojimi lastnostmi natančneje opiše dejavni projekt (znotraj razvojnega okolja) oziroma izvajajoči se program (če gre za samostojen program). Na voljo sta tudi dve metodi, ki omogočita zapis *dnevnika dogodkov* (*event log*).

Lastnosti

Lastnost **Title**

Morda najuporabnejša lastnost predmeta App, ki opisuje naziv projekta. Določimo ga lahko na dva načina:

- V projektnem oknu izberemo sam projekt in v oknu za nastavljanje lastnosti spremenimo lastnost Name.
- Izberemo ukaz **Project ➪ Urejevalnik options...** (ime ukaza je odvisno od trenutnega naziva projekta). V prikazanem pogovornem oknu izberemo jeziček **General** in vnesemo ime v polje **Project Name**.

Ta nastavitev odseva v lastnosti Title. Dobro je, če je ime projekta in poznejšega programa opisno in *enolično* za vse naše projekte. Zaradi prenosljivosti kode se na to lastnost pogosto sklicujemo na različnih mestih:

```
' Branje / nastavljanje nastavitev v registru. V ta namen mora biti naziv
' enoličen, da ne prekrijemo/preberemo nastavitev druge aplikacije!
GetSetting(App.Title, "Settings", "MainLeft", 1000)
```

```
' Izpis naziva projekta v modalnem pogovronem oknu.
MsgBox "Grozljiva napaka", vbCritical, App.Title & " - Napaka"

' Preverjanje / nastavljanje izvora napake.
Err.Raise vbObjectError + 1, App.Title, "Nova napaka"
```

In še kje.

Lastnosti **Path** in **EXEName**

Lastnost Path vrne pot, na kateri je izvajajoči se program. V razvojnem okolju je to pot do projektne datoteke (.VBP), pri samostojnem programu pa gre za pot do datoteke .EXE. Njeno ime podaja lastnost EXEName. Z naslednjima stavkoma določimo, da je dejavna pot ob zagonu programa enaka poti, na kateri je naš program:

```
Private Sub Form_Load()
  ChDir App.Path
  ChDrive App.Path
End Sub
```

Lastnost **PrevInstance**

Lastnost PrevInstance vrne logično vrednost, ki ponazarja hkratno izvajanje več izvodov istega programa. Njena vrednost je resnična, če se hkrati izvaja vsaj še en izvod. Naslednji zgled prepreči izvajanje v več izvodih:

```
Private Sub MDIForm_Initialize()
  If App.PrevInstance Then
    ' dopušča se le en izvod!
    End
  End If
End Sub
```

Za tako opravilo je najprimernejši dogodek Initialize. V njem lahko za konec dela brez posebne škode uporabimo stavek End.

Predmet App ima odgovarjajočo lastnost še za vsak parameter programa, ki ga lahko nastavimo v lastnostih projekta (glej zgoraj, Prevajanje). Nekatere izmed njih smo uporabili za izdelovanje vizitke. Njihov natančnejši opis je na voljo v sprotni pomoči. Tu si oglejmo še dve zanimivi lastnosti:

Lastnosti **LogPath** in **LogMode**

Programi v Oknih 95 in NT lahko pomembnejše dogodke zapisujejo v dnevnik (*event log*). Dnevnik je v Oknih NT bolje sistemsko podprt, saj je zanj pripravljena posebna datoteka in uslužnostni pregledovalnik. Z lastnostjo LogPath lahko določimo pot do datoteke z dnevnikom. Lastnost LogPath natančneje opisuje način zapisovanja dogodkov. Naslednji zgled je uporaben na obeh platformah. Nastavitev lastnosti LogMode določa, da se v oknih NT uporabi sistemski dnevnik, v Oknih 95 pa datoteka, na katero kaže nastavitev LogPath:

```
App.LogPath = App.Path & "\DNEVNIK.LOG"
App.LogMode = vbLogAuto ' v Oknih NT izberi sistemski dnevnik
```

Zapis dogodkov je mogoč z metodo **LogEvent**.

Metodi

Metoda **StartLogging**

Z metodo StartLogging lahko na mah postorimo to, kar določata lastnosti LogPath in LogMode skupaj. Prej zapisani zgled bi lahko določili tudi takole:

```
App.StartLogging App.Path & "\DNEVNIK.LOG", vbLogAuto
```

Metoda **LogEvent**

Metoda LogEvent doda nov zapis v dnevnik. Sprejme dva parametra: niz z opisom dogodka in kodo njegove vrste. Ta koda je lahko ena izmed pripravljenih konstant:

Konstanta	Vrsta dogodka
vbLogEventTypeError	Napaka
vbLogEventTypeWarning	Opozorilo
vbLogEventTypeInformation	Informacija

Z uporabo predmeta App izvemo marsikaj o našem programu. Za povrh nam ponuja še možnost trajnega zapisa pomembnejših dogodkov v *dnevnik*, ki je na voljo tudi med samostojnim izvajanjem programa.

> **NAMIG** V razvojnem okolju lahko za izpisovanje sporočil, ki ne motijo izvajanja programa, vedno uporabimo tudi predmet Debug. Vendar je njegova vloga omejena na razvojno okolje in se v prevedenem programu ne upošteva več.

Uporaba funkcij Windows API

Če se ozremo po menujih Urejevalnika, bomo opazili še en ukaz, ki ga doslej nismo obdelali. To je ukaz Razveljavi v menuju Urejanje. Razveljavitev zadnjega ukaza bi lahko sprogramirali v visual basicu. Potrebovali bi še en niz, ki bi hranil kopijo besedila pred zadnjo spremembo. Na željo uporabnika bi ta niz prestavili v vnosno polje.

Na voljo pa je še lažja pot. Večvrstično vnosno polje je zelo zmogljiv gradnik, ki nam lahko skoraj popolnoma nadomesti Beležnico (*Notepad*). Žal pa nam načrtovalci visual basica niso dovolili neposrednega dostopa do vseh zmožnosti, ki jih vnosno polje pozna. Razveljavitev zadnje spremembe je namreč v vnosno polje že vgrajena, le da ni na voljo nobena lastnost ali metoda, ki bi nam omogočala dostop do nje.

Kar ni pripravljeno, si večinoma lahko pripravimo sami. Pri tem nam pomagajo funkcije Windows API (*Application Programmer Interface*). Te predstavljajo programerski vmesnik, ki ga uporabljajo vsi okenski programi. V njih je zbrana vsa funkcionalnost operacijskega sistema, ki jim je na voljo. Veliko večino teh funkcij lahko uporabimo tudi v visual basicu.

Osnovna težava pri uporabi funkcij API v visual basicu je, da ni ustrezne dokumentacije. Tudi najboljši visual basic (*Enterprise Edition*) nima nobene dokumentacije o funkcijah API, razen pripravljenih *napovedi*. Da bi funkcijo API lahko koristno uporabili, moramo poznati:

- Ime funkcije in njen namen.
- Zahtevane parametre in njihov podatkovni tip.
- Pomen vrnjene vrednosti in njen podatkovni tip.

Ustrezno dokumentacijo najdemo v paketu Visual Studio ali v kakem drugem razvojnem okolju.

Preden lahko izbrano funkcijo API neposredno pokličemo iz kode, jo moramo ustrezno *napovedati*. Napoved zunanje funkcije dosežemo s stavkom **Declare**, ki ga moramo zapisati v del modula s splošnimi napovedmi. Znotraj obrazcev lahko napovemo le modulsko dosegljive funkcije, zato je boljše mesto za zapis napovedi samostojen modul s kodo, kjer so napovedi lahko globalne in neposredno dostopne kjerkoli v programski kodi. Popolna oblika stavka Declare je:

```
[Public|Private] Declare Function ime Lib "knjižnica" [Alias "sinonim"]
[([parametri])] As tip
```

Vsi podprogrami Windows API so funkcije, vendar lahko večino napovemo tudi kot proceduro, če nas vrnjena vrednost ne zanima:

```
[Public|Private] Declare Sub ime Lib "knjižnica" [Alias "sinonim"]
[([parametri])]
```

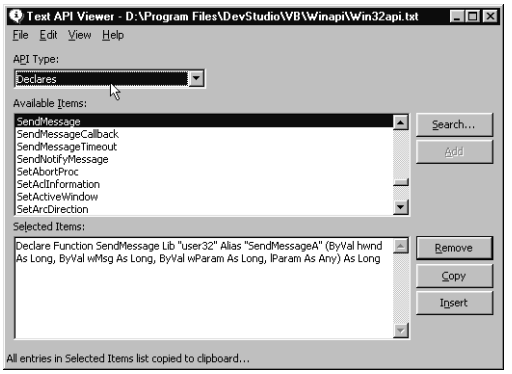
Pri zapisu napovedi nam pomaga orodje Text API Viewer, ki že vsebuje napovedi za vse dostopne okenske funkcije (teh je kar 1527) in z njimi povezane konstante (6370) ter podatkovne strukture (414 različnih).

Za uporabo zmožnosti razveljavitve bomo potrebovali funkcijo SendMessage. Njeno napoved najdemo takole:

1. Izberemo ukaz **API Viewer** iz glavnega menuja **Add-Ins**.

Požene se pomožni program Text API Viewer, ki rabi kot pregledovalnik napovedi, konstant in tipov, shranjenih v zunanji datoteki. Pred uporabo moramo zato najprej naložiti ustrezno datoteko.

Slika 9-19: Pripomoček API Viewer



2. V menuju **File** izberemo ukaz **Load Text File...**
3. Znotraj mape »Winapi« (je v mapi, v kateri je nameščen visual basic) izberemo datoteko »Win32api.txt«.
4. Preverimo, ali je na seznamu **API Types** izbrana vrstica Declarations.
5. Na seznamu **Available Items** bi morala biti imena razpoložljivih funkcij.
6. Kliknemo gumb **Search**, v vnosno polje vnesemo »sendmessage« in pritisnemo Enter.
7. Na seznamu je izbrana funkcija SendMessage. Njeno napoved prepišemo v vnosno polje **Selected Items** s klikom gumba **Add**.
8. Vsebino seznama Selected Items z gumbom **Copy** prepišemo na odložišče.
9. Pregledovalnik napovedi je opravil svoje delo, zato ga lahko zapremo (**File ➪ Exit**) in se vrnemo v visual basic.
10. Napoved moramo zapisati znotraj podrejenega obrazca frmBesedilo. Prikažemo njegov kodni modul in v delu splošnih napovedi iz priročnega menuja (ali iz glavnega menuja **Edit**) izberemo ukaz **Paste**.

Prikaže se stavek Declare, ki mu ročno dodamo še pridevnik Private:

```
Private Declare Function SendMessage Lib "user32" Alias … ' se nadaljuje
```

Funkcija SendMessage omogoča dostop do najrazličnejših zmožnosti vgrajenih gradnikov. Razveljavitev zadnjega ukaza v vnosnem polju dosežemo zelo preprosto:

```
' Razveljavitev / uveljavitev zadnjega vnosa
Public Sub Razveljavi()
Const EM_UNDO = &HC7
  SendMessage txtBesedilo.hWnd, EM_UNDO, 0, 0
End Sub
```

Funkciji moramo posredovati posebno vrednost, enolično oznako okna[35] oziroma njegovo ročico. Skoraj vsak gradnik pozna lastnost **hWnd** (*window handle*), ki je ravno ta oznaka. Sporočilu, ki vnosnemu polju ukazuje, naj razveljavi zadnji ukaz, pripada koda, določena s konstanto EM_UNDO. Vrednost te konstante bi prav tako lahko izvedeli iz pregledovalnika Text API Viewer, če bi na seznamu API Types namesto Declarations izbrali Constants (glej zgoraj opisani postopek). Funkcija SendMessage sicer prejme še dva parametra, ki pa morata biti za uspešno razveljavitev zadnjega ukaza enaka nič.

> **NAMIG** Pregledovalnik Text API Viewer brska po običajnih datotekah z besedilom. Če ga pogosto uporabljamo, bomo vedno kar precej časa čakali, da bo prebral vse napovedi. Zato nam po dolgotrajnih branjih sam ponudi možnost pretvorbe besedila v zbirko podatkov. To možnost lahko kadarkoli izberemo tudi sami. Potem ko smo naložili želeno datoteko z besedilom (**File ➪ Load Text File**...), jo pretvorimo v zbirko podatkov z ukazom **File ➪ Convert Text To Database**... Izberemo še ime ciljne datoteke, in to je vse.
>
> Zbirka podatkov sicer zavzame približno dvakrat več prostora na disku, vendar se naloži bistveno hitreje. Žal pa uporaba zbirke podatkov prinese hudo nevšečnost. Iz kdo ve katerega razloga pri uporabi zbirke podatkov ni več na voljo gumb **Search**, ki omogoča preiskovanje!?

Klic funkcije API smo skrili v javno metodo obrazca, ki jo zdaj lahko pokličemo iz kode glavnega obrazca:

```
Private Sub mnuUrejanjeRazveljavi_Click()
  ActiveForm.Razveljavi
End Sub
```

V Urejevalniku lahko izkoristimo še eno opravilo, ki ga v vnosnem polju zna opraviti funkcija SendMessage. Ker že imamo pripravljeno vrstico stanja, bi jo lahko izkoristili še za izpisovanje položaja vnosnega kazalca. Ugotavljanje vrstice in stolpca znotraj vnosnega polja pa je vse prej kakor lahko delo. Brez funkcije SendMessage skorajda ne bi šlo.

---

[35] Spomnimo se, da je za programerja »okno« vsako pravokotno območje zaslona, ki lahko sprejema sporočila. Vrednost ročice okna se med izvajanjem programa lahko spremeni, zato je ne smemo prepisovati v spremenljivko.

Vrstico stanja smo že doslej obnavljali v zasebni proceduri ObnoviStanje. Z uporabo funkcije SendMessage jo lahko razširimo do naslednje oblike:

```
Private Sub ObnoviStanje()
Dim intVrstica As Integer, intStolpec As Integer
Const EM_LINEFROMCHAR = &HC9
Const EM_LINEINDEX = &HBB
  intVrstica = SendMessage(txtBesedilo.hwnd, EM_LINEFROMCHAR, _
    txtBesedilo.SelStart, 0&)
  intStolpec = txtBesedilo.SelStart - SendMessage(txtBesedilo.hwnd, _
    EM_LINEINDEX, intVrstica, 0&)
  lblStatus.Caption = IIf(blnSprememba, "(Spremenjeno)", "") & _
     "      Vr " & Format(intVrstica + 1, "000") & ", St " & _
     Format(intStolpec + 1, "000") & "      "
End Sub
```

Za ugotavljanje trenutne vrstice in stolpca, v katerem je kazalec vnosa (*caret*), uporabimo dva klica funkcije SendMessage. Vse, kar poznamo, je zaporedna številka znaka, pred katerim je kazalec, ki ga vrne lastnost SelStart. S prvim klicem funkcije SendMessage neposredno ugotovimo vrstico, ki pripada znani zaporedni številki znaka. Z drugim klicem izvemo zaporedno številko *prvega* znaka v pravkar najdeni vrstici. Še nekaj matematike in že imamo izračunan stolpec. Ker se vrednosti za vrstico in stolpec začneta z 0, ju pred izpisom še povečamo za 1. Oblikovan izpis dosežemo s funkcijo Format, za lepši vtis pa na desni strani vrinemo še nekaj praznega prostora.

Vrstico stanja je sedaj treba mnogo pogosteje obnavljati. Zato moramo zapisati še dve odzivni proceduri:

```
Private Sub txtBesedilo_KeyUp(KeyCode As Integer, Shift As Integer)
  ObnoviStanje
End Sub

Private Sub txtBesedilo_MouseUp(Button As Integer, Shift As Integer, _
X As Single, Y As Single)
  ObnoviStanje
End Sub
```

Dogodka KeyDown in MouseDown nista primerna, saj se pripetita še *pred* premikom kazalca. Zgoraj uporabljena dogodka se pripetita po tem, ko je kazalec že premaknjen, zato sta pravo mesto za obnovitev vrstice stanja.

Toliko za uvod. Zanimive funkcije Windows API bomo spoznavali pri reševanju novih programerskih težav.

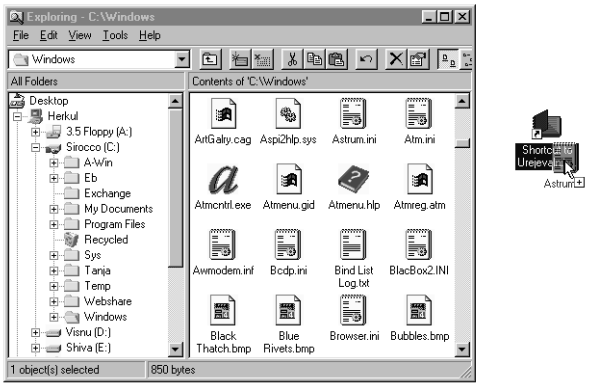
NAMIG S stavkom **Declare** lahko napovemo tudi zunanjo funkcijo iz katerekoli deljene knjižnice (datoteke DLL), ki je dostopna med izvajanjem programa. Praviloma so dostopne datoteke, ki so v isti mapi kakor izvršljivi program; v osnovni ali sistemski mapi Oken ali v eni od map, vsebovanih v sistemski nastavitvi PATH.

Vlogo zunanjih funkcij v deljenih knjižnicah so v zadnjem času skorajda v celoti prevzeli zunanji gradniki ActiveX, ki so prilagodljivejši in jih je mogoče mnogo laže uporabiti.

Branje ukazne vrstice

Če smo Urejevalnik prevedli v samostojni program, ga lahko uporabimo za koristna opravila. Brez težav lahko na namizje dodamo njegovo bližnjico, da ga imamo vedno hitro pri roki. Naredimo kratek preizkus.

1. Najprej na namizju (*desktop*) ustvarimo bližnjico do Urejevalnika.
2. Z Raziskovalcem (*Explorer*) najdimo kako datoteko z besedilom.
3. Z miško zvlecimo datoteko nad ikono bližnjice Urejevalnika in jo izpustimo.
4. Urejevalnik se požene, vendar je njegovo glavno okno prazno.



Slika 9-20: Zagon urejevalnika prek namizja

Kaj bi morali storiti, da bi Urejevalnik znal odpreti nanj odvržene datoteke? Opravilo je preprostejše, kakor se zdi.

Vsak program lahko poenostavljeno obravnavamo tudi kot »podprogram«, ki ga uporablja glavni program: operacijski sistem. Ob zagonu vsakega programa lahko ta prejme še parametre, podobno kakor pravi podprogrami. Tem parametrom pravimo tudi *parametri ukazne vrstice* (*command line arguments*). To ime velja še danes, saj imajo skoraj vsi grafični vmesniki posebno ukazno okno (*console*, *DOS prompt* itn.), v katerem lahko programe ročno poganjamo in jim posredujemo dodatne parametre. Mehanizem parametrov ukazne vrstice izdatno uporablja tudi operacijski sistem, le da je ta proces za uporabnika neviden.

V visual basicu lahko parametre ukazne vrstice preberemo s funkcijo **Command**. Ta naenkrat vrne *vse* morebitne parametre v obliki niza znakov. Za preverjanje posredovanja parametrov ni nujno, da program prevedemo v samostojno obliko in ga zatem ročno poganjamo iz ukazne vrstice. Preizkusne vrednosti ukaznih parametrov lahko nastavimo v izvajalnem okolju, skupaj z drugimi nastavitvami projekta. Izberemo ukaz **Urejevalnik properties...** (ime ukaza je odvisno od imena projekta) v glavnem menuju **Project**. V pogovornem oknu izberemo jeziček **Make**. Prikazana stran vsebuje vnosno polje z oznako **Command Line Arguments**, kamor lahko vnesemo preizkusne vrednosti ukaznih parametrov.

V našem zgledu bomo funkcijo Command uporabili v prvem odzivnem podprogramu, ki se pokliče ob začetku izvajanja programa. To je procedura MDIForm_Load v glavnem obrazcu. Njena posodobljena oblika je (nove vrstice so označene krepko):

```
Private Sub MDIForm_Load()
Dim strParam As String, strDatoteka As String

  Caption = App.Title
  With dlgCommonDialog
    .Filter = "Besedilo (*.txt)|*.txt|Vse datoteke (*.*)|*.*"
    .DefaultExt = ".txt"
  End With

  strParam = Command
  Do While NaslednjaBeseda(strParam, strDatoteka)
    NovoOkno strDatoteka
  Loop

  ' Vzpostavimo menuje
  OmogociMenuje
End Sub
```

Ker so vsi parametri zbrani v enem samem nizu, bomo morali razviti še posebno funkcijo, ki bo iz podanega niza poizkušala izločiti naslednjo besedo, njena vrnjena vrednost pa bo ponazarjala uspešnost tega opravila. Ker je prav mogoče, da v ukazni vrstici ni nobenega parametra, moramo uporabiti obliko zanke Do... Loop s preverjanjem pogoja na vrhu, ki se lahko tudi nikoli ne izvede v celoti.

Funkcija NaslednjaBeseda je silno uporabna v različnih projektih, kjerkoli uporabljamo *členjenje* niza. Členjenje (*parsing*) je postopek, v katerem niz razdelimo na njegove sestavne dele, ki jih pozneje smiselno uporabimo, npr. za ugotavljanje posredovanih datotek. Zaradi široke uporabnosti naslednje funkcije jo bomo uvrstili v neodvisen modul s kodo. Njen izpis je:

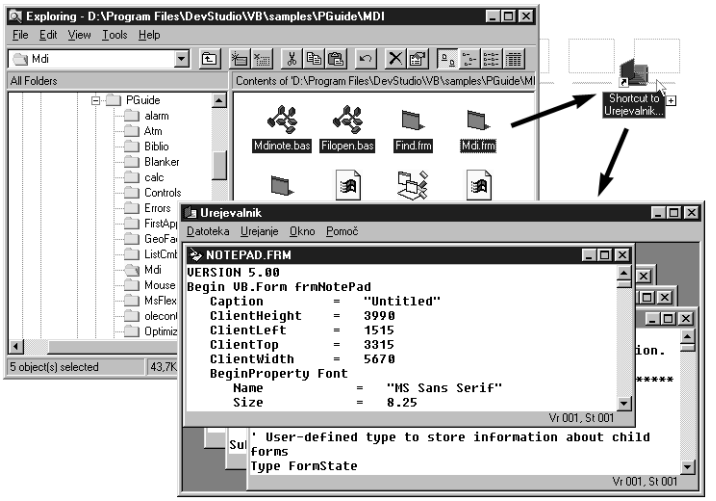
```
Public Function NaslednjaBeseda(ByRef strNiz As String, _
  ByRef strDat As String) As Boolean
Dim znak As String * 1
Dim blnNiz As Boolean
Dim i As Integer
```

```
  If Len(strNiz) > 0 Then
    strDat = ""
    blnNiz = False
    strNiz = LTrim(strNiz)
    For i = 1 To Len(strNiz)
      znak = Mid(strNiz, i, 1)
      If znak = """" Then blnNiz = Not blnNiz
      If Not blnNiz And (znak = " " Or znak = vbTab) Then
        Exit For
      Else
        strDat = strDat & znak
      End If
    Next
    strNiz = Mid(strNiz, Len(strDat) + 1)
    NaslednjaBeseda = Len(strDat)
  Else
    NaslednjaBeseda = False
  End If
End Function
```

Pri funkciji je bistveno, da sta parametra posredovana po referenci. Čeprav je to privzeti način posredovanja, smo ga eksplicitno zapisali zaradi lažjega razumevanja delovanja. Funkcija tako dejansko »vrne« kar tri vrednosti, saj posodobi vrednosti posredovanih parametrov. Če bi tega ne storila, bi se prej zapisana zanka nikoli ne iztekla.

Osnovni pogoj delovanja funkcije je, da ima sploh s čim delati. Vgrajena funkcija Len vrne dolžino posredovanega niza. Naslednja uporabljena funkcija, LTrim, iz niza odstrani ves morebitni *beli prostor* na levi strani (*whitespace*, skupni izraz za nevidne znake, kot so: presledki, znaki za zamik in prehod v novo vrsto itn.). Visual basic pozna še sorodni funkciji RTrim in Trim, ki odstranita beli prostor z desne oziroma z obeh strani niza. Delo nadaljujemo tako, da se v zanki For sprehodimo po celotnem preostalem nizu. Pri tem moramo paziti na poseben znak, dvojni narekovaj ("). Ker so lahko v dolgih imenih datotek tudi presledki, hkrati pa je presledek tudi ločilo posameznih parametrov ukazne vrstice, je treba dolga imena zapisati med narekovaji. Zanka se lahko predčasno konča, če naletimo na beli prostor, ki ni v nizu. Vgrajena konstanta vbTab označuje kodo znaka za zamik. Na koncu popravimo vrednost posredovanega niza, tako da bomo z zaporednimi klici iste funkcije drugega za drugim izluščili vsak parameter. Uspešnost funkcije je odvisna od dolžine izluščenega podniza.

Zdaj lahko z večjimi pričakovanji ponovimo zgled, opisan na začetku obravnave ukazne vrstice. Če bomo na ikono bližnjice do Urejevalnika odvrgli poljubno število označenih datotek iz Raziskovalca (*Explorer*) ali podobnega programa, bo Urejevalnik vsak primeren dokument odprl v novem oknu.



Slika 9-21: Odpiranje datotek z bližnjico in Raziskovalcem

NAMIG Celotno kodo Urejevalnika najdete na priloženi plošči CD-ROM v mapi: \Primeri\2-Delavnica\2-Urejevalnik\4-Koda. Uporaba vmesnika MDI je nakazana tudi pri programu MDINote, ki spremlja visual basic, njena izvorna koda pa je v mapi Samples\PGuide\MDI.

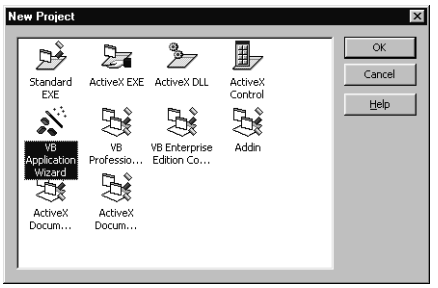
Čarovnije

Spoznavanje razvoja uporabniškega vmesnika je prav gotovo ena temeljnih vaj pri učenju uporabe visual basica. A če bi morali ob vsakem novem projektu ponavljati dolgočasne korake vzpostavljanja vmesnika, bi nas veselje do razvijanja programov hitro minilo. Delo nam lahko olajša uporaba predlog. Visual basic ponuja še zmogljivejši in enostavnejši način vzpostavitve programa, ki se imenuje Application Wizard. Pravzaprav gre za prilagodljivo predlogo, ki nas skozi nekaj pogovornih oken pripelje do značilnih uporabniških vmesnikov z nekaterimi že izdelanimi elementi. Za tovrstne pripomočke se je uveljavilo ime »čarovnik«.

Čarovnik Application Wizard je na voljo kot ena izmed predlog v galeriji, ki se prikaže na začetku novega projekta, bodisi z ukazom **File ➪ New...** bodisi z ukazom **File ➪ Add Project...** Njegova ikona je vidna na sliki 9-22.

Ko ga izberemo, nas pričaka prvo iz zaporedja osmih pogovornih oken, ki nas usmerjajo do želenega izhodišča novega programa. Vsak trenutek lahko pregledamo naslednje ali prejšnje okno (z gumboma **Next** in **Back**) ali pa prepustimo čarovniku, naj delo dokonča namesto nas (gumb **Finish**).

Slika 9-22: Čarovnik Application Wizard je ena izmed predlog novega projekta.

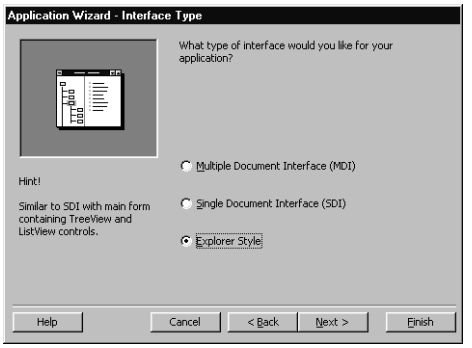


Prvo okno čarovnika je zgolj uvodni pozdrav. Njegovo bodoče prikazovanje lahko izključimo z vklopom stikala **Skip this screen in the future**.

Slika 9-23: Prvo, pozdravno okno čarovnika Application Wizard

V drugem koraku izberemo obliko uporabniškega vmesnika. Na voljo imamo vmesnike MDI, SDI in posnemanje Raziskovalca (**Explorer Style**). Slednja dva sta si podobna v tem, da obdelujeta le en dokument, vendar pri posnemanju Raziskovalca čarovnik samodejno doda še nekaj zunanjih gradnikov ActiveX, ki povzemajo značilnosti vmesnika, prvič predstavljenega v Oknih 95. Najznačilnejše bomo spoznali v naslednjem poglavju.

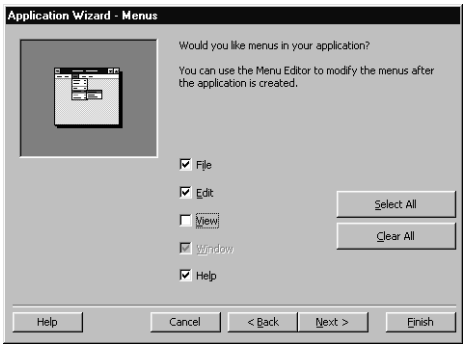
Slika 9-24: Najprej izberemo želeni uporabniški vmesnik.



Naslednji korak omogoča samodejno dodajanje nekaj značilnih menujev, kot so:

- File, v katerem so ukazi za delo z datotekami.
- Edit, ki vsebuje ukaze za urejanje.
- View, ki vsebuje ukaze za vklop in izklop določenih elementov vmesnika, npr. vrstice stanja.
- Window, ki je na voljo samo pri vmesniku MDI, vsebuje pa ukaze za izbiranje in urejanje podrejenih oken.
- Help, ki vsebuje ukaze za priklic sprotne pomoči in vizitke.

NAMIG Čeprav so menuji ponazorjeni z angleškimi imeni, je mogoče v naslednjem koraku določiti, da se njihovi napisi berejo iz zunanje datoteke sredstev. Z njo je mogoče imena menujskih ukazov, pa tudi druge nize, izpisane na vmesniku, enostavno prilagoditi drugemu jeziku.



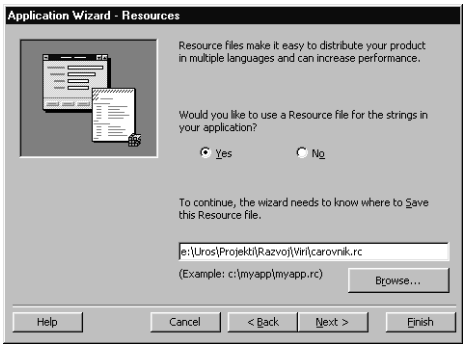
Slika 9-25: Sledi izbor želenih menujev.

V naslednjem koraku določimo, ali želimo besedilo vmesnika shraniti v zunanjo *datoteko sredstev*. Datoteka sredstev (*resource file*) lahko vsebuje različne elemente uporabniškega vmesnika, kot so nizi, bitne slike, kazalci miške, zvok, animacije in podobno. Prednost uporabe datoteke sredstev je v njeni ločenosti od programske kode. Čeprav se vsebina te datoteke ob prevajanju zapiše v isto datoteko kakor izvršilna koda programa (izvršljiva datoteka vrste .EXE), jo je mogoče enostavno nadomestiti z novo vsebino, brez vnovičnega prevajanja in brez kakršnegakoli vpliva na programsko kodo.

Če izberemo stikalo **Yes**, moramo določiti še ime zunanje datoteke sredstev. Pri imenu moramo natančno opisati pot, sicer bo datoteka s sredstvi shranjena v dejavno mapo, ki je lahko kjerkoli na disku. Pri tem opravilu nam lahko pomaga gumb **Browse...**, ki prikaže standardno pogovorno okno za izbiro datotek. Čarovnik bo v to datoteko zapisal vsebino vseh nizov, ki se pojavljajo na uporabniškem

vmesniku (npr. imena menujskih ukazov), zato je poznejše prilagajanje vmesnika zelo olajšano. Zunanjo datoteko sredstev lahko enostavno pripravimo in uporabimo tudi v vseh drugih projektih v visual basicu. To opravilo bomo podrobno spoznali v naslednjem poglavju.

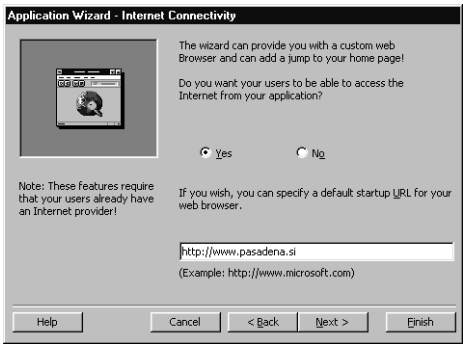
Slika 9-26: Za lažje prilagajanje vmesnika lahko uporabimo zunanjo datoteko sredstev.



Korak, ki sledi, je morda celo odveč. No, če želite v vaš projekt vgraditi tudi spletni brskalnik, ni nobenih ovir. S pritrdilnim odgovorom bo vaš projekt pestrejši za dva zunanja gradnika ActiveX, ki omogočita deskanje po spletu zgolj z nastavitvijo nekaj lastnosti. Za delovanje brskalnika seveda potrebujete delujočo povezavo z internetom.

NAMIG Vklop te zmožnosti je priporočljiv za spoznavanje z ustreznimi zunanjimi gradniki. Čarovnik bo skupaj z gradniki v projekt dodal tudi vso potrebno kodo.

Slika 9-27: V projekt lahko preprosto vključite tudi spletni brskalnik.



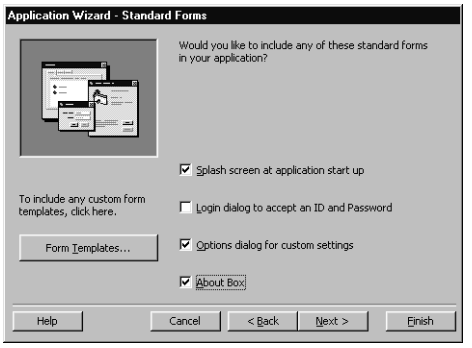
Naslednje okno ponuja nekaj dodatnih možnosti, ki jih pogosto potrebujemo pri najrazličnejših projektih (no, nekateri so tudi bolj ali manj v okras). Na voljo so:

- Splash Screen – uvodno pozdravno okno, ki se prikaže ob nalaganju programa.

- Login dialog – pogovorno okno, ki zahteva vnos uporabniškega imena in gesla ob začetku dela. To možnost vklopimo pri projektih, pri katerih uporabnik dela z zaupnimi podatki. Geslo lahko uporabimo tudi za poznejši priklop na podatkovni strežnik (delo s podatkovnimi strežniki bomo spoznali v četrtem delu knjige).
- Options dialog – večstransko pogovorno okno za najrazličnejše nastavitve. Prikaže se ob izboru ustreznega menujskega ukaza (v angleški različici je to **View ➪ Options...**).
- About Box – pripravljena vizitka (predlogo, na kateri temelji, smo že omenili).

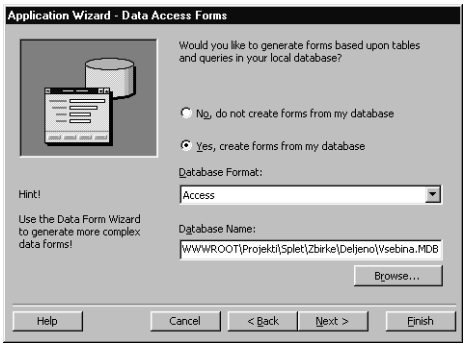
Z gumbom **Form Templates...** lahko v projekt vključimo še poljubno število obrazcev, temelječih na pripravljenih predlogah (uporabimo lahko le predloge iz mape »Forms«).

Slika 9-28: Nekaj dodatnih možnosti projekta



V predzadnjem koraku lahko v projekt dodamo še več obrazcev za obdelovanje podatkov v izbrani *zbirki podatkov*. Če bomo določili podatkovni vir, bomo morali določiti še *tabele* oziroma *poizvedbe*, na katerih bodo temeljili novi obrazci. Tudi ta korak je v čarovniku nekoliko odveč, saj nam celo sam svetuje, da za bolj zapletene obrazce raje uporabimo posebnega čarovnika. Kot že rečeno, bomo zbirke podatkov, pa tudi omenjenega posebnega čarovnika, spoznali v četrtem delu knjige.

Slika 9-29: S čarovnikom je mogoče hitro dodati tudi preproste obrazce za obdelovanje zbirk podatkov.



Zadnji korak zahteva še ime celotnega projekta. Z vklopom stikala **Yes** nam čarovnik izdela tudi poročilo z opisom opravil, ki jih bo treba na izdelanem temelju še postoriti.

Pri uporabi čarovnika lahko v kateremkoli koraku kliknemo gumb **Finish**, in temelj projekta se bo samodejno pripravil do konca. Če želimo v prihodnje pri teh samodejnih pripravah uporabiti nastavitve, ki smo jih pravkar določili za nastajajoči program, vklopimo še stikalo **Save current settings as default**.

Vaje

Tudi urejevalnik je mogoče izboljšati in ob tem utrditi pridobljeno znanje. Tu je nekaj nalog:

1. Prestavite vrstico stanja v glavno okno vmesnika MDI. Vedno naj odraža stanje v trenutno dejavnem oknu.
2. Pri branju parametrov ukazne vrstice odpremo toliko novih oken, kolikor je posredovanih datotek. Izbor več datotek nam omogoča tudi standardno pogovorno okno za odpiranje. Predelajte Urejevalnik tako, da bo omogočal izbor več datotek in bo vsako odprl v novem oknu (nasvet: več datotek je mogoče v standardnem pogovornem oknu izbrati s pritisnjeno tipko Ctrl ali Shift, povsem enako kakor pri razširjenem izbiranju s seznama, seveda pa ga je treba najprej omogočiti).
3. Če že uporabljamo standardna pogovorna okna, dodajmo še možnost izbiranja pisave, ki se uporabi za izpis besedila. Ne potrebujemo novega gradnika, saj obstoječi pokriva vsa standardna pogovorna okna. Poizkusite omogočiti različno pisavo za vsako podrejeno okno.
4. V glavni menu Urejanje dodajmo več uslužnostnih funkcij. Tu je nekaj predlogov:
   - Pretvorba označenega besedila v velike/male črke.
   - Zamenjava želenega števila presledkov z znakom za zamik (tab) in nasprotno.
   - Štetje števila označenih ali vseh znakov z belim prostorom ali brez njega.
5. Dober urejevalnik besedila mora omogočati izpis s tiskalnikom. Nadzoru tiskalnika iz visual basica je posvečen uvodni del naslednjega poglavja. Ko ga preučite, dodajte Urejevalniku možnost izpisa na poljuben tiskalnik. Tudi izbiro tiskalnika in njegove nastavitve je mogoče prikazati z dvema standardnima pogovornima oknoma.

10. POGLAVJE

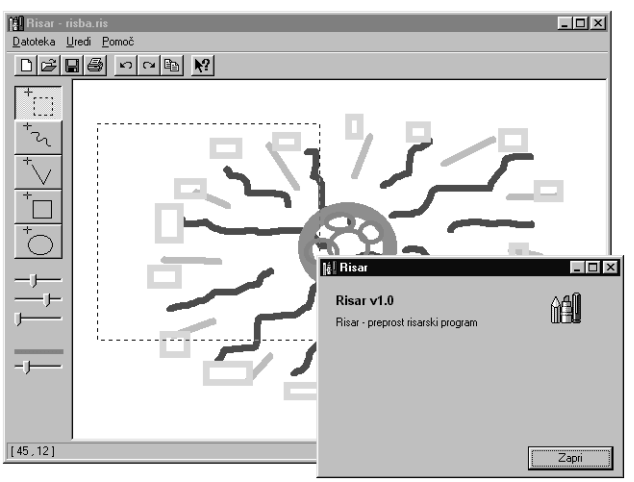
Risar

Zadnji zgled v drugem delu knjige se bo ukvarjal predvsem z grafiko. Poleg tega bomo spoznali še nekaj zelo pomembnih dopolnil, kot je ustvarjanje in uporaba datotek s sredstvi (*resource files*) in razvoj sprotne pomoči (*on-line help*). Preden se lotimo dela, si znova pregledno oglejmo vse novosti, ki jih bomo spoznali:

- Uporaba grafike, koordinatni sistem in barve.
- Gradniki za delo z grafiko.
- Grafične lastnosti in metode.
- Vgrajena predmeta Screen in Printer.
- Razvoj vmesnika z elementi Oken 95.
- Izdelovanje in uporaba datotek s sredstvi.
- Nizkonivojsko delo z miško.
- Delo z binarnimi datotekami.
- Opremljanje programa s sprotno pomočjo.
- Razpečevanje izdelanega programa.

Videz delujočega programa si lahko ogledate na spodnji sliki. Izvorna koda zgledov je na priloženi plošči CD-ROM, v mapah znotraj mape \Primeri\2-Delavnica\3-Risar\.

Slika 10-1: Risar med delom



Uporaba grafike

Okna so operacijski sistem z *grafičnim* uporabniškim vmesnikom (GUI, *Graphical User Interface*). Ta vmesnik programom omogoča, da uporabljajo najrazličnejše grafične elemente, kot so gumbi, stikala, menuji, ikone, kazalci miške itn. Povrhu lahko vsak program izrisuje tudi lastno grafiko, z grafičnimi lastnostmi in metodami, pri tem pa uporablja najrazličnejše pisave in obravnava vse zunanje enote zelo podobno (zaslon, tiskalnik, risalnik itn.). V visual basicu imamo za oblikovanje vmesnika na voljo dovolj izdelanih gradnikov. Včasih pa želimo dodati tudi kak nov grafični element. Pri tem imamo na izbiro dvoje:

- Uporabimo grafični gradnik.
- Grafiko izrišemo programsko, z lastnostmi in metodami.

Med obema načinoma je kup pomembnih razlik. Že iz dosedanjih zgledov vemo, da nam za *grafično* podobo gradnikov ni treba skrbeti, saj se znajo sami upodobiti v vsakem trenutku. Posebni grafični gradniki – spoznali jih bomo v nadaljevanju – so prav tako sposobni skrbeti sami zase. Uporaba grafičnih metod pa lahko prinese nekaj zapletov. Preden spoznamo uporabo obeh načinov dodajanja lastne grafike, si oglejmo nekaj lastnosti, ki bistveno vplivajo na njuno rabo.

Koordinatni sistemi

Koordinatni sistem seveda uporabljamo že od vsega začetka. Kar spomnimo se lastnosti Left in Top, ki ju pozna vsak gradnik. Vendar se doslej še nismo vprašali, kaj njuni nastavitvi dejansko pomenita in kako se izmerita. Čas je, da natančneje spoznamo koordinatne sisteme, ki so na voljo v Oknih.

Koordinatne sisteme delimo v dve skupini, po:

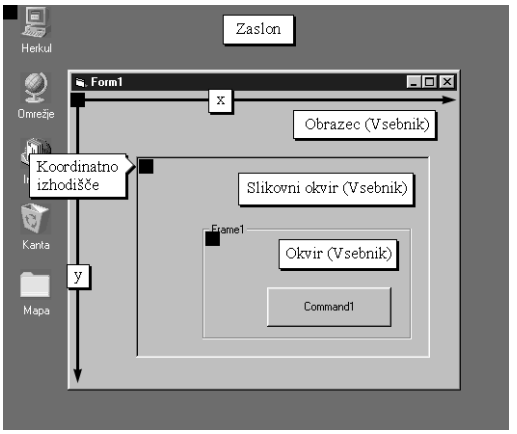
- njihovem vplivu in
- njihovi skali.

Po vplivu jih delimo na zaslonski koordinatni sistem, tiskalnikov koordinatni sistem in vsebniške koordinatne sisteme. Zaslonski koordinatni sistem pripada celotnemu zaslonu in določa položaj glavnih oken. Njegove skale ni mogoče spremeniti in se vedno meri v posebni merski enoti, *twipih*. Da bi ga laže obvladovali, nam pomaga vgrajeni predmet Screen. Tudi tiskalnik pozna svoj koordinatni sistem, vendar je ta precej bolj prilagodljiv. Nadziramo ga z vgrajenim predmetom Printer. Vsebniški koordinatni sistemi pripadajo vsakemu gradniku, ki je vsebnik (*container*). Doslej smo spoznali naslednje vsebnike:

- Vsaka vrsta obrazca.
- Okvir (*Frame*).
- Slikovni okvir (*Picture Box*).

Vsebniški koordinatni sistemi vplivajo na vse gradnike in grafične metode, ki jih znotraj njih uporabimo. Po tej značilnosti bi lahko med vsebnike prišteli tudi ves zaslon. Vsebniški koordinatni sistem vedno opisuje zgolj delovno površino znotraj vsebnika (*client area*) in ne njegovih fizičnih dimenzij. Te so določene v koordinatnem sistemu njegovega nadrejenega vsebnika, npr. zaslona, z lastnostmi Left, Top, Width in Height. Več različnih koordinatnih sistemov in njihovih privzetih izhodišč prikazuje naslednja slika:

Slika 10-2: Koordinatni sistemi v Oknih



Vsem koordinatnim sistemom, razen zaslonskemu, lahko predpišemo *skalo*. Skala (*scale*) določa izhodišče in razpon razpoložljivih koordinat. Pri njenem določanju imamo na voljo nekaj pripravljenih merskih sistemov, ki se opirajo na fizične dimenzije, merski sistem, prilagojen zaslonu, in uporabniško določen koordinatni sistem. S slednjim lahko spremenimo tudi izhodišče koordinatnega sistema, ki je sicer vedno v levem zgornjem vogalu. Skalo v celoti določimo z eno ali petimi lastnostmi. Pri vgrajenih skalah zadošča nastavitev lastnosti **ScaleMode**. Pozna jo vsak vsebnik in predmet Printer. Možne nastavitve so zbrane v spodnji tabeli:

Lastnost ScaleMode	Vgrajena konstanta	Skala
0 – User	vbUser	Uporabniška skala, določena z dodatnimi lastnostmi.
1 – Twip	vbTwips	Posebna skala v izmišljeni merski enoti *twip* (privzeta skala).
2 – Point	vbPoints	Tiskarske pike (pika = 1/72 palca).
3 – Pixel	vbPixels	Zaslonske pike.
4 – Character	vbCharacters	Znaki.
5 – Inch	vbInches	Palci.

Lastnost ScaleMode	Vgrajena konstanta	Skala
6 – Millimeter	vbMillimeters	Milimetri.
7 – Centimeter	vbCentimeters	Centimetri.

Natančneje si oglejmo pomen posameznih nastavitev.

Twipi, točke, pike in palci

Večina vnaprej pripravljenih skal, razen uporabniške skale in zaslonskih točk, temelji na fizičnih merah, izraženih v različnih merskih enotah. Privzeta merska enota je *twip*. To je izmišljena merska enota, določena kot 1/1440 palca, oziroma 1/567 centimetra. Zaradi take definicije je uporabna v različnih merskih sistemih, saj natančno določa razmerje med palci in centimetri – mimogrede, to znaša približno 2,54 cm/palec. Tudi tiskarske pike temeljijo na palcu, in sicer predstavljajo njegovo 1/72, kar je enako 20 twipom. Posebna skala so znaki (4 – Character), ki imajo različno vrednost za vodoravno (120 twipov na enoto) in navpično os (240 twipov na enoto). Vse te skale na zaslonu niso preveč uporabne, nam pa pridejo nadvse prav za izpisovanje s tiskalnikom.

Za zaslon je najuporabnejša skala, izražena v zaslonskih pikah ali *pikslih* (*pixels*, prirejena kratica za *Picture Element*). Zaslonska pika je najmanjša enota zaslona, ki nima posebne zveze s fizično enoto, čeprav je to razmerje ves čas natančno določeno (izvemo ga lahko s predmetom Screen).

> **NAMIG** To razmerje je potrebno zaradi enakega obravnavanja različnih izhodnih naprav, kot sta npr. zaslon in tiskalnik. Seveda je povsem razumljivo, da 100 zaslonskih pik ne moremo predstaviti s fizično mero, saj imamo zaslone najrazličnejših velikosti, vsak pa omogoča prikaz več različnih ločljivosti.

Ko spremenimo nastavitev lastnosti ScaleMode, se samodejno pretvorijo vrednosti vseh lastnosti, ki opisujejo položaj in velikost gradnikov (Left, Top, Width in Height).

Uporabniška skala

Pri nastavitvah skale imamo na voljo še štiri lastnosti in posebno metodo **Scale**. Lastnosti so naštete v naslednji tabeli:

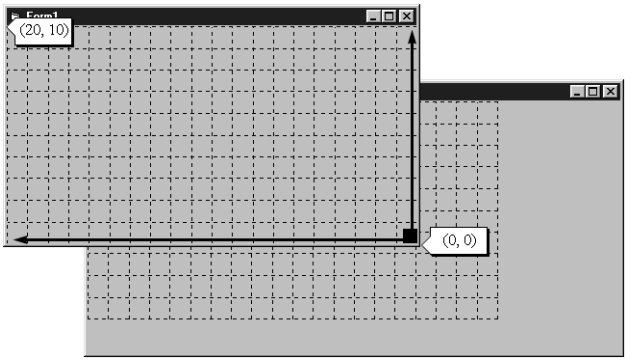
Lastnost	Opis
ScaleTop	Določa novo višino koordinatnega izhodišča.
ScaleLeft	Določa novo širino koordinatnega izhodišča.
ScaleWidth	Določa razpon širine v enotah skale.
ScaleHeight	Določa razpon višine v enotah skale.

Vse lastnosti so realna števila enojne natančnosti (vrsta Single). Zadnji dve lastnosti nam pomagata pri ugotavljanju skrajnih razsežnosti koordinatnega sistema. Uporabili smo ju npr. pri Urejevalniku, za dinamično spreminjanje položaja in velikosti gradnikov. Kakor hitro katero izmed naštetih lastnosti nastavimo na novo vrednost, vklopimo uporabniški koordinatni sistem. Lastnost ScaleMode samodejno privzame vrednost 0. Vse lastnosti hkrati lahko hitreje spremenimo z metodo Scale, ki kot parameter sprejme dva koordinatna para: zgornji levi vogal in spodnji desni vogal.

S kombiniranjem različnih vrednosti lahko določimo najrazličnejše koordinatne sisteme. Naslednja nastavitev:

```
With Form1
   .ScaleLeft = 20
   .ScaleTop = 10
   .ScaleWidth = -.ScaleLeft
   .ScaleHeight = -.ScaleTop
End With
```

popolnoma zasuka orientacijo obeh osi, ker sta oba razpona podana kot negativni vrednosti. Koordinatno izhodišče je sedaj desno spodaj, vrednosti pa se povečujejo vodoravno levo proti 20 in navpično navzgor proti 10. Ta koordinatni sistem se določi glede na trenutno razsežnost vsebnika. Če se ta pozneje spremeni (npr. ko razširimo obrazec), je treba uporabniški koordinatni sistem vnovič nastaviti. To prikazuje tudi naslednja slika:



Slika 10-3: Uporabniško določena skala

Enako stanje bi lahko dosegli s klicem metode Scale, ki prejme koordinati obeh oglišč:

```
Form1.Scale (20, 10)-(0, 0)
```

Klic metode Scale brez parametrov izbere privzeti koordinatni sistem in skalo.

Oglejmo si še en zanimiv in zelo uporaben zgled, ki zasuka pripravljeno skalo, denimo tisto v centimetrih:

```
' Nov koordinatni sistem
Printer.ScaleMode = vbCentimeters
' preberemo širino in višino v centimetrih
sngVisina = Printer.ScaleHeight
sngSirina = Printer.ScaleWidth
' zasukamo koordinatni sistem
Printer.Scale (0, sngVisina)-(sngSirina, 0)
```

S tem smo na papirju dobili zelo uporabno skalo. Merska enota se ujema s centimetri, koordinatno izhodišče pa je levo spodaj, kakor smo vajeni.

Za pretvorbo med različnimi merskimi enotami imamo na voljo dve funkcijski metodi, **ScaleX** in **ScaleY**, ki vrneta razmerje podanega števila enot v dveh različnih skalah. Naslednji zgled:

```
Form1.ScaleWidth = 10
MsgBox Form1.ScaleX(3, vbUser, vbHimetric)
```

Izpiše, koliko stotink centimetra pripada trem desetinam uporabniško določene širine obrazca. Konstante vbHimetric nismo zasledili med uporabniškimi skalami. Pri pretvorbi imamo poleg vseh skal na voljo še tri posebne vrednosti:

Vrednost	Vgrajena konstanta	Opis
8	vbHimetric	Stotinke centimetra.
9	vbContainerPosition	Določa položaj gradnika v vsebniku.
10	vbContainerSize	Določa velikost gradnika v vsebniku.

Trenutni položaj

Skala neposredno vpliva še na dve pomembni lastnosti: **CurrentX** in **CurrentY**. Ti dve lastnosti opisujeta izhodišče naslednje grafične operacije. V začetku sta enaki koordinatnemu izhodišču. Pozneje se prilagajata spremembam skale, njuna vrednost pa se ustrezno spremeni tudi po klicu vsake grafične metode. Po želji ju lahko tudi ročno nastavljamo.

Barve

Tudi z barvami se srečujemo že od prvega srečanja z lastnostjo BackColor. Pri načrtovanju barve praviloma izbiramo s sistemske ali splošne palete in morda niti nismo opazili, da je vsaka barva predstavljena z dolgim celim številom (tip Long). V oknu za nastavljanje lastnosti je ta vrednost ponazorjena v šestnajstiškem zapisu. Splošna oblika tega zapisa je:

```
&HPPRRGGBB
```

Mesta RR, GG in BB označujejo intenzivnost rdečega, zelenega in modrega odtenka v končnem barvnem tonu. Dovoljeni razpon je v šestnajstiškem zapisu med 00 in FF, kar da 256 kombinacij oziroma skupaj 16,7 milijona različnih barv (256^3). Mesti PP označujeta posebno oznako, ki vpliva na pomen drugih vrednosti. Pri mešanju barv mora biti ta oznaka &H00. Če je na tem mestu npr. koda &H80, preostala mesta ne opisujejo več intenzivnosti odtenka, ampak zaporedno številko barve na sistemski paleti. Barva, ki pripada površini gumba, je vedno 15. barva na sistemski paleti oziroma barva s kodo:

```
&H8000000F
```

Seveda bomo barve le redko kodirali s številom. Večinoma uporabimo enega izmed dveh načinov:

- Barvo s sistemske palete določimo z vgrajeno konstanto.
- Svojo barvo namešamo s funkcijo **RGB**.

Za uporabniški vmesnik, ki se sklada z drugimi programi, moramo uporabljati barve s sistemske palete. Tako bo naš program prevzel sistemske nastavitve in bo najbolj spoštoval želje uporabnika[36]. Oglejmo si zgled:

```
' rumeno ozadje
BackColor = RGB(255, 255, 0)
' barva ospredja je enaka barvi gumba
ForeColor = vbButtonFace
```

Palete

Čeprav je mogoče namešati kar 16,7 milijona različnih barv, je dejansko prikazana barva določena z zmožnostmi strojne opreme. Če ima uporabnik grafično kartico, ki podpira prikaz samo 16 barv, bodo dosledno prikazane le barve, izbrane s sistemske palete, vse druge pa bodo upodobljene s tehniko *rastriranja*. Rastriranje (*dithering*) je postopek, v katerem določene barve ponazorimo z različno gostoto pik na osnovni barvni podlagi. Če npr. ni mogoče prikazati oranžne barve, jo ponazorimo z vzorcem rdečih pik, posejanih po rumeni podlagi.

[36] Uporabnik lahko sistemsko barvno paleto spremeni v sistemskih nastavitvah.

Do rastriranja pride tudi pri 256-barvni paleti (8-bitna barvna globina), v kateri so poleg sistemskih lahko še dodatne barve, ki pa jih je treba nekako določiti. Na vsebino palete in rastriranje barv vplivata dve lastnosti *obrazca*: **Palette** in **PaletteMode**.

Lastnost PaletteMode določa način, kako se uporabljene barve prilagodijo ali preslikajo na sistemsko paleto. Nastavitev te lastnosti je posebej pomembna pri vmesnikih, ki uporabljajo več slik s številnimi barvami in se izvajajo v sistemih z 256-barvnim prikazom. Največkrat rabljene nastavitve so:

Lastnost Palette Mode	Opis
vbPaletteModeHalfTone	Uporabi privzeto paleto in rastrira neobstoječe barve (privzeta nastavitev).
vbPaletteModeUseZOrder	Uporabi paleto, ki pripada vrhnjemu gradniku. Ta nastavitev lahko povzroči nadležno preklapljanje palete.
vbPaletteModeCustom	Uporabi paleto, določeno z lastnostjo Palette.

Privzeta nastavitev pogosto povsem zadovoljuje, saj rastriranje dovolj dobro ponazori vsako barvo. Včasih imamo za vse obrazce natančno določeno paleto in nam bolj ustreza tretja možnost. Preden jo uporabimo, moramo lastnosti Palette prirediti bitno sliko, ki vsebuje 256-barvno paleto. Ta slika lahko vsebuje eno samo točko, saj je pomembna zgolj njena paleta. Oglejmo si zgled:

```
Form1.Palette = LoadPicture(App.Path & "\pal256.bmp")
Form1.PaletteMode = vbPaletteModeCustom
```

Celoten uporabniški vmesnik uporablja le barve, ki so vsebovane v datoteki pal256.bmp. Spotoma smo spoznali še zelo uporabno funkcijo **LoadPicture**, ki – kakor vidimo – prebere poljubno bitno sliko, shranjeno v zunanji datoteki v kateremkoli od podprtih oblik zapisa, vključno z GIF in JPEG.

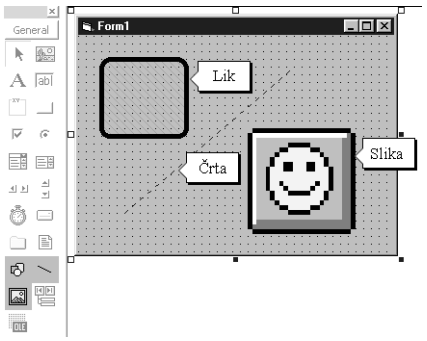
> **NAMIG** Če v programu ne uporabljamo veliko barv, lahko nastavitev vbPaletteModeCustom celo malo pospeši prikaz obrazca, saj v tem primeru ne pride do preslikovanja barvnih palet.

Pri višjih barvnih ločljivostih (15-, 16-, 24- ali 32-bitna globina barv) palete in rastriranja ni več[37].

[37] Prav vse barve se predstavijo z najbližjo izmed razpoložljivih barv. V večini primerov da to povsem dobre rezultate, opaznejši barvni trakovi se lahko pojavijo le pri množični uporabi odtenkov iz ozkega dela barvnega spektra (npr. širok preliv iz temno modre v črno barvo).

Grafični gradniki

Kakor smo že omenili, lahko uporabniškemu vmesniku dodajamo lastno grafiko z grafičnimi metodami ali grafičnimi gradniki. Gradnike je nekoliko laže uporabljati, vendar so manj prilagodljivi in zahtevnejši do sistemskih sredstev. Bistvena prednost uporabe gradnikov je, da so vidni že med načrtovanjem in njihovega izrisovanja ni treba posebej nadzorovati.



Slika 10-4: Grafični gradniki

Med grafične gradnike prištevamo lik (*Shape*) in črto (*Line*). Posebna vrsta grafičnega gradnika je še preprosta shramba slik, gradnik slika (*Image*), ki smo ga delno že spoznali.

Lastnosti

Lastnosti **X1**, **Y1**, **X2**, **Y2**

Samo črta, zaradi lažje uporabe, ne pozna lastnosti Left, Top, Width in Height, temveč določimo njen položaj z dvema paroma koordinat.

Lastnost **Shape**

Lastnost Shape določa enega izmed geometrijskih likov, ki jih lahko ponazori gradnik lik (*Shape*). Na voljo imamo pravokotnik (0 – Rectangle) in kvadrat (1 – Square), po želji z zaobljenimi vogali (4, 5), ter elipso (2 – Oval) in krog (3 – Circle).

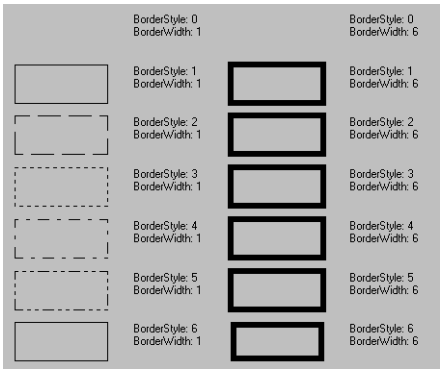
Lastnosti **BorderStyle**, **BorderWidth** in **BorderColor**

Naštete lastnosti vplivajo na videz obrobe lika (*Shape*). Pri črti je to bolj ali manj tudi vse, pri likih pa moramo poleg obrobe določiti tudi notranjost. Lastnost BorderStyle določa slog črte, ki je lahko poln, pikčast, črtkan, kombinacija pik in črt, odsoten ali vsebovan. Črtkani in pikčasti slogi delujejo le, če je debelina obrobe – to določa lastnost BorderWidth – enaka 1! Barvo obrobe določa lastnost BorderColor. Posebej velja omeniti prosojni in vsebovani slog črte obrobe. Pri prosojnem slogu se obroba pač ne vidi, lahko pa s svojo debelino zoži notranjost lika. Vsebovani slog črte

pomeni, da se celotna debelina črte izriše znotraj površine lika, sicer se obroba izriše pol znotraj, pol zunaj.

Vplive različnih nastavitev prikazuje spodnja slika:

Slika 10-5: Slog obrobe lika

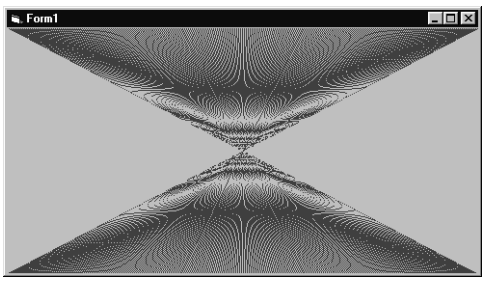


Nastavitev sloga v programski kodi je nazornejša, če uporabimo katero od pripravljenih konstant (npr. vbBSDashDot za črtkano-pikčasto obrobo).

Lastnost **DrawMode**

Lastnost DrawMode določa bitno operacijo, ki se izvede med že prisotnimi pikami podlage in novo izrisanimi pikami v črti ali liku. Na voljo je vseh 16 različnih bitnih operacij, ki omogočajo doseganje posebnih učinkov, kot je ta na spodnji sliki:

Slika 10-6: Posebni učinki z lastnostjo DrawMode



Najuporabnejša nastavitev je prav gotovo vbXorPen (uporabljena je bila tudi v zgornjem zgledu), ki omogoča odstranitev novo izrisane vsebine brez vpliva na podlago. Izdatno jo bomo uporabljali v primeru, ki ga bomo razvili pozneje v tem poglavju.

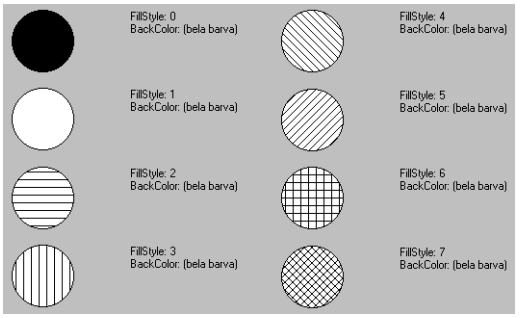
Lastnosti **BackStyle** in **BackColor**

Lik lahko po želji zapolnimo z izbrano barvo, ki jo določa lastnost BackColor. Da bo ta vidna, moramo nastaviti lastnost BackStyle na vrednost 1 – Opaque. Po privzeti nastavitvi je notranjost likov prosojna.

Lastnosti **FillStyle** in **FillColor**

Ne glede na barvno zapolnitev lahko vsebino lika še dodatno zapolnimo z eno izmed pripravljenih šrafur. Njen videz določa lastnost FillStyle, barvo pa lastnost FillColor. Nekatere kombinacije zapolnitev so prikazane na spodnji sliki:

Slika 10-7: Vpliv zapolnitev



S tem smo opisali pomembne lastnosti grafičnih gradnikov črta in lik. Preostanejo nam še lastnosti gradnika slika (*Image*).

Lastnost **Picture**

Lastnost Picture smo že omenili. Določa sliko, ki je shranjena v gradniku. Lastnost lahko nastavimo na več načinov:

- Preberemo vsebino datoteke s funkcijo LoadPicture.
- Priredimo ji grafično vsebino iz datoteke sredstev.
- Priredimo ji lastnost Picture drugega gradnika Image ali kakega drugega grafičnega vsebnika (obrazca ali slikovnega okvira).
- Priredimo ji lastnost Icon obrazca.
- Priredimo ji bitno sliko, ki je shranjena na odložišču (*clipboard*).

Če v gradnik naložimo sliko že med načrtovanjem, se bo ta shranila tudi v poznejšo izvršljivo datoteko programa (.EXE). Zaradi te lastnosti je bil gradnik Image nekoč najbolj učinkovit način shranjevanja slik. Zdaj so na voljo učinkovitejše metode, kot je uporaba datoteke sredstev ali posebnega zunanjega gradnika. Obe možnosti bomo spoznali pri razvoju zgleda v tem poglavju.

NAMIG Če želimo že včitano sliko odstraniti iz gradnika, moramo med načrtovanjem s tipko Del zbrisati nastavitev, med izvajanjem pa je sliko najlaže odstraniti s posebnim klicem funkcije LoadPicture:

```
Image1.Picture = LoadPicture("")
```

Lastnost **Stretch**

Po privzeti nastavitvi se velikost gradnika samodejno prilagodi velikosti prirejene slike. Če želimo ravno nasprotno vedenje: širjenje ali krčenje prikazane slike glede na določene velikosti gradnika, nastavimo lastnost Stretch na resnično vrednost (*true*).

Metode

Pri delu z grafičnimi gradniki ni posebnih metod, ki jih še ne bi spoznali. Zelo pomembna je ZOrder, s katero lahko med izvajanjem določamo grafični gradnik, ki je na vrhu oziroma na dnu. Ker so grafični gradniki na posebnem *sloju* (*layer*), jih ne moremo postaviti povsem v ospredje, npr. pred ukazni gumb, urejamo lahko le njihovo medsebojno urejenost. O slojih bomo izvedeli več v nadaljevanju.

Dogodki

Lik in črta ne poznata dogodkov, slika pa je omejena in ne more prejeti vnosne točke, zato ne pozna dogodkov, povzročenih s tipkovnico.

Grafične lastnosti in metode

Grafični gradniki nam že med izvajanjem nakazujejo svoj videz, za svojo podobo skrbijo sami in za to porabijo le minimalno sistemskih sredstev. Kljub temu ni učinkovito razviti npr. risarskega programa, v katerem bi vsako izrisano črto ali geometrijski lik ponazorili s pripadajočim gradnikom. Nekaterih likov, denimo krožni izsek, z gradnikom sploh ni mogoče izrisati. Grafične metode uporabimo pri vseh zahtevnejših izrisovanjih, pri katerih bi za podoben učinek potrebovali npr. deset ali več gradnikov.

Seveda imajo grafične metode tudi slabost. Predvsem moramo za njihov učinek program pognati, neredko pa je treba za obnavljanje izrisane grafike posebej skrbeti. Za enostavne, dekorativne posege na obrazcu zato raje uporabimo gradnik ali dva.

> **NAMIG** Gradniki nam pridejo prav tudi pri preizkušanju vpliva določenih lastnosti (npr. nastavitve zapolnitve), ki jih pozneje uporabimo skupaj z grafičnimi metodami.

Uporabi grafičnih lastnosti in metod pravimo v žargonu *izrisovanje* grafike (*painting*).

Kje izrisovati?

Z grafičnimi lastnostmi in metodami lahko rišemo znotraj grafičnih vsebnikov, kot sta obrazec in slikovni okvir (gradnik PictureBox). Ker je v okolju Oken tudi tiskalnik zgolj še ena izhodna naprava, lahko z grafičnimi metodami rišemo tudi na predmet Printer, končno sliko pa na koncu odtisnemo na papir, oziroma izrišemo na mediju, ki ga uporablja izhodna naprava (to je lahko tudi risalnik, rezalnik, telefaks itn.).

Lastnosti

Vsak predmet, na katerega lahko izrisujemo lastno grafiko z grafičnimi metodami, pozna nekaj lastnosti, ki vplivajo na poznejše izrisovanje. Večino smo že spoznali pri dosedanjem obravnavanju grafike, življenje nam greni le spremenjeno ime nekaterih. Zato si jih pregledno oglejmo v naslednji tabeli:

Lastnost	Opis
Lastosti Scale...	Določajo koordinatni sistem in skalo.
Lastnost BackColor	Določa barvo ozadja grafičnega vsebnika. Nastavitev te lastnosti hkrati pobriše vso dotlej izrisano vsebino.
Lastnosti CurrentX in CurrentY	Določata izhodišče naslednjega izrisovanja oziroma konec prejšnjega.
Lastnost ForeColor	Podobna je lastnosti BorderColor, le da vpliva na barvo obrobe pri izrisovanju z grafičnimi metodami, hkrati pa določa tudi barvo izrisanega besedila.
Lastnosti DrawWidth in DrawStyle	Enakovredni lastnostma BorderWidth in BorderStyle, le da vplivata na izrisovanje z grafičnimi metodami. Torej določata debelino in slog obrobe.
Lastnost DrawMode	Določa bitno operacijo med podlago in novo izrisano grafiko.
Lastnost FillColor in FillStyle	Določa barvo in slog zapolnjevanja likov.

Na obstojnost in vedenje grafičnih metod bistveno vplivata še dve lastnosti, ki pa ju bomo prihranili za pozneje. Sedaj izrišimo nekaj lastne grafike...

Metode

Vse izrisovalne metode grafičnih vsebnikov in predmeta Printer so zbrane v naslednji tabeli:

Metoda	Opis
Cls	Zbriše izrisano grafiko.
PSet	Obarva izbrano piko v želeno barvo.
Point	Vrne barvo izbrane pike.
Line	Izriše črto ali pravokotnik.
Circle	Izriše elipso, lok ali izsek.
Print	Izriše besedilo.

Precej pisana, a zelo uporabna druščina. Oglejmo si jo podrobneje.

Metoda **Cls**

Izrisane grafike ne moremo posebej spreminjati, lahko jo le prerišemo ali zbrišemo. Metoda Cls zbriše vso izrisano grafiko in grafični vsebnik obravnava z barvo ozadja, določeno z lastnostjo BackColor. Na grafične in druge gradnike nima nikakršnega vpliva.

Metoda **PSet**

Popolna oblika metode PSet je:

```
[predmet.]PSet [Step](x,y)[, barva]
```

Za grafične metode v visual basicu je značilno, da jim podajamo koordinatni par v oklepaju, ki ga po želji opremimo še z rezervirano besedo Step. Če jo izpustimo, sta podani koordinati *absolutni*, drugače pa koordinatni par določa zgolj *odmik* od trenutnega položaja, določenega z lastnostma CurrentX in CurrentY. Vsako izrisovanje seveda popravi ti dve lastnosti na novo vrednost:

```
Scale (0, 0)-(20, 20)
PSet (10, 10)   ' => CurrentX=10, CurrentY=10
PSet Step(-5, 5) ' == PSet(5,15) => CurrentX=5, CurrentY=15
```

Pri metodi PSet izberemo piko, ki jo bomo obarvali v barvo ospredja (po lastnosti ForeColor) ali v naknadno določeno barvo s tretjim, neobveznim parametrom. Za brisanje pike uporabimo barvo ozadja:

```
With Form1
  .PSet (10, 10), .BackColor
End With
```

Na izrisovanje vpliva še lastnost DrawWidth, ki določa debelino pike, in DrawMode, ki določa bitno operacijo z obstoječo podlago.

Metoda **Point**

Metoda Point vrne barvo izbrane pike. Če je ta barva drugačna od barve ozadja (lastnost BackColor), je na tem mestu izrisana grafika. Metoda vrne negativno vrednost –1, če sta podani koordinati zunaj obsega vsebnika.

Zgled uporabe, ki zbriše morebiti obarvano piko:

```
With Form1
  If .Point(10, 10) <> .BackColor Then .PSet (10, 10), .BackColor
End With
```

Metoda **Line**

Popoln zapis metode Line je kar zapleten:

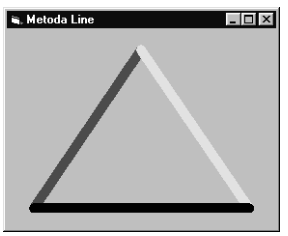
```
[predmet.]Line [[Step](x1, y1)]-[Step](x2, y2)[, barva][, B[F]]
```

Metoda poveže dve piki s črto ali pa med njima izriše (zapolnjen) pravokotnik. Če prvi koordinatni par izpustimo, se uporabi trenutni položaj (po lastnostih CurrentX in CurrentY). Poleg barve lahko kot zadnji parameter zapišemo črko B (kot *box*, škatla), ki ukaže risanje pravokotnika namesto črte, ali zaporedje BF (kot *box*, *filled*, zapolnjena škatla), ki ukaže izris zapolnjenega pravokotnika. Pri tem se za obrobo *in* zapolnjevanje uporabi podana barva ali lastnost ForeColor (če barvo izpustimo). Precej čudno, vendar smo to podedovali iz starejših različic basica. Na izrisovanje obrobe vplivajo lastnosti DrawStyle, DrawWidth in DrawMode, pri risanju pravokotnika brez zapolnjevanja (samo B) pa tudi FillStyle in FillColor, ki določata zapolnitev.

Ker lahko z zaporednimi klici metode Line izrišemo lomljene črte ali večkotnike, se pri izrisovanju črte *ne* obarva pika, ki jo določa drugi koordinatni par[38]. Oglejmo si zgled izrisovanja trikotnika:

```
DrawWidth = 10
Scale (0, 0)-(20, 20)
' prva stranica je opisana z obema koordinatama
Line (2, 18)-Step(8, -16), vbRed
' druga stranica je opisana le z relativnim odmikom
Line -Step(8, 16), vbYellow
' na koncu trikotnik zaključimo v izhodišču
Line -(2, 18), vbBlack
```

Slika 10-8: Izrisani trikotnik s tremi klici metode Line



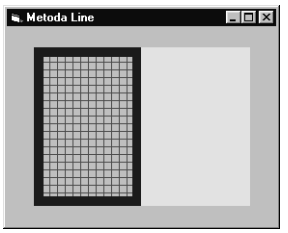
Še zgled zapolnjenih pravokotnikov:

```
DrawStyle = 6
DrawWidth = 10
```

[38] Težave bi se lahko pojavile pri posebnih nastavitvah lastnost DrawMode, kot je npr. vbXorPen, kjer bi dvakratno barvanje iste pike to pobrisalo. S tem bi oglišča ostala neobarvana.

```
Scale (0, 0)-(20, 20)
FillStyle = 6
FillColor = vbRed
' Pravoktnik uporabi nastavitve lastnosti Fill...
Line (2, 18)-Step(8, -16), vbBlue, B
' Zapolnjeni pravokotnik uporabi le podano barvo
Line -Step(8, 16), vbYellow, BF
```

Slika 10-9: Zapolnjeni pravokotniki



Metoda **Circle**

Metoda Circle je prav tako večnamenska. Z njo lahko izrišemo kroge, elipse, krožne loke ali izseke. Popolna oblika zapisa je:

[*predmet*.]**Circle** [**Step**](*x*, *y*), *radij* [, *barva*, *začetek*, *konec*, *razmerje*]

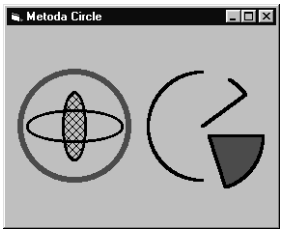
Koordinatni par podaja središče, obvezni parameter radij pa polmer. V tej osnovni obliki bo metoda izrisala krog v obrobi, določeni z lastnostmi ForeColor, DrawWidth, DrawStyle in DrawMode. Po želji lahko barvo obrobe eksplicitno zapišemo. Začetek in konec podajata kot v radianih, med katerimi se izriše krožni lok ali izsek. Z razmerjem lahko krog spremenimo v elipso. Razmerje, manjše od 1, splošči krog navpično, večje od ena pa ga zoži vodoravno. Čeprav lahko z lastnimi koordinatnimi sistemi povsem spremenimo razmerja med vodoravno in navpično osjo, pomena razmerij osi elipse ne moremo spremeniti. Enka (ali izpuščeni parameter) vedno predstavlja popoln krog. Za mersko enoto radija se vedno uporabi skala vodoravne osi, osi x.

Ali bo metoda izrisala krožni lok ali krožni izsek, je odvisno od predznaka obeh kotov, ki sta podana kot parametra za začetek in konec. Negativne vrednosti se *ne* tolmačijo kot kot, merjen v negativni smeri, ampak zgolj ponazarjajo zahtevo po povezavi oglišč loka s središčem. Oglejmo si zgled:

```
Dim pi As Double
' Izračunamo Pi
pi = 4 * Atn(1)

Scale (0, 0)-(40, 20)
DrawWidth = 6
' Popoln krog
Circle (10, 10), 8, vbRed
DrawWidth = 3
' Navpično sploščena elipsa
Circle Step(0, 0), 7, , , , 1 / 3
```

```
FillStyle = vbDiagonalCross
FillColor = vbBlack
' Vodoravno zožana elipsa
Circle Step(0, 0), 5, , , , 3
' Zapolnitev za zaključene izseke
FillStyle = vbFSSolid
FillColor = vbRed
' Krožni lok
Circle Step(19, 0), 8, , pi / 2, 3 * pi / 2
' Le delno zaključen izsek
Circle Step(0, 0), 8, , -pi / 5, pi / 3
' Povsem zaključen izsek se še zapolni
Circle Step(1, 1), 8, , -8 * pi / 5, -0.00001
```

Slika 10-10: Različne uporabe metode Circle



Pri zaključevanju izseka v kotu 0 moramo zapisati zelo majhno negativno število, ker se –0 ne razume kot negativna vrednost (verjetno zaradi hrošča).

NAMIG Preračunavanje stopinj v radiane je mogoče po naslednji enačbi:

```
<kot v radianih> = <kot v stopinjah> * Pi / 180
```

Vsi koti se vedno merijo v pozitivni smeri, to je smer, nasprotna gibanju urnega kazalca. Visual basic ima na voljo pravo zakladnico trigonometričnih funkcij, v sprotni pomoči pa so z njimi izražene tudi mnoge druge.

Metoda **Print**

Metodo Print smo sicer že spoznali, ko smo obravnavali predmet Debug. Z grafično metodo Print ju druži le podoben namen in skladnja, izvedba pa je povsem drugačna. Grafična metoda Print izrisuje besedilo v izbrani barvi in pisavi znotraj grafičnega vsebnika ali na predmet Printer. Njena popolna oblika je:

```
[predmet.]{Print|?} [seznamZaIzpis]
```

Pri tem je seznam za izpis sestavljen iz poljubnega števila naslednjih elementov:

```
{Spc(n)|Tab[(n)]|izraz|[{,|;}]}
```

Kakor vidimo, lahko ukaz okrajšamo samo z vprašajem (ki ga bo urejevalnik samodejno pretvoril v metodo Print). Na izpisnem seznamu lahko nastopajo poljubni izrazi, saj jih metoda pred izpisom pretvori v niz, podobno kakor funkcija

CStr (glej pretvorbo tipov v prvem delu). Po želji lahko izpis zamaknemo s funkcijama Spc, ki vstavi želeno število presledkov, oziroma Tab, ki prestavi izpis za določeno število stolpcev. Funkcijo Tab lahko pokličemo tudi brez parametra, tedaj se predpostavi vrednost 14. Med elementi seznama ali na koncu lahko uporabljamo dva posebna simbola:

- Podpičje (;) pomeni, da se bo naslednji izpis začel tik za zadnjim.
- Vejica (,) pomeni, da se bo naslednji izpis začel z zamikom, enakim funkciji Tab brez posredovanega parametra.

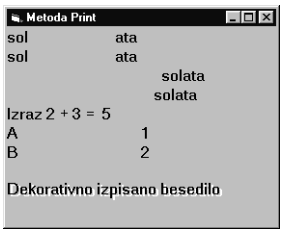
Če končnega simbola ne zapišemo, napravi metoda Print prehod v novo vrstico. Pred vsakim izpisovanjem lahko položaj izpisa natančno nastavimo z lastnostma CurrentX in CurrentY.

> **NAMIG** Nastavitev stolpcev s funkcijo Tab ne deluje pravilno pri sorazmernih pisavah, kjer je npr. črka »m« mnogo širša od črke »i«. Za poravnavo besedila lahko uporabimo nesorazmerno pisavo, kot je »Courier New«, ali pa določamo stolpec neposredno z lastnostjo CurrentX.

Oglejmo si zgled najrazličnejših klicev metode:

```
Scale (0, 0)-(100, 100)
Font.Size = 12
Print "sol"; Tab; "ata"
Print "sol", "ata"
Print Spc(20); "solata"
Print Tab(20); "solata"
Print "Izraz 2 + 3 = "; 2 + 3
Print "A";
CurrentX = 50
Print "1"
Print "B";
CurrentX = 50
Print "2"
Print ' samo prehod v novo vrsto
' Senca
ForeColor = vbWhite
CurrentX = 1
Print "Dekorativno izpisano besedilo"; ' <- podpičje!
' Še enkrat preko v črni barvi in zamaknjeno
CurrentX = 0
CurrentY = CurrentY - 1
ForeColor = vbBlack
Print "Dekorativno izpisano besedilo"
```

Kakor vidimo, na izpis vplivajo še nastavitve pisave (lastnost Font) in barva ospredja (ForeColor).



Slika 10-11: Izris besedila z metodo Print

Lastnost Font lahko zaradi združljivosti s staro kodo določamo na dva načina:

```
' Po starem…
FontName = "Arial"
FontSize = 20
FontBold = True

' Po novem…
With Font
  .Name = "Arial"
  .Size = 20
  .Bold = True
End With
```

Oba načina sta enakovredna, vendar je priporočljivejši drugi, saj morda prvi ne bo več dolgo podprt.

Poleg izrisovalnih metod poznajo grafični vsebniki in predmet Printer še nekaj uslužnostnih:

Metodi **TextWidth** in **TextHeight**

Metodi izračunata širino oziroma višino, ki *bi* jo zasedel posredovani niz, izpisan v izbrani pisavi. Z njima lahko npr. ugotovimo, ali bo naslednjo besedo še mogoče v celoti izpisati v trenutno vrstico ali pa izračunamo enoinpolkratni presledek med vrsticami. Naslednji zgled bo uporabil obe metodi za to, da bo niz izpisal natančno na sredini obrazca:

```
Dim strNiz As String
strNiz = "Besedilo v popolni sredini"
CurrentX = (ScaleWidth - TextWidth(strNiz)) \ 2
CurrentY = (ScaleHeight - TextHeight(strNiz)) \ 2
Print strNiz
```

Metoda **PaintPicture**

Grafični uporabniški vmesniki morajo pogosto premikati obširna pravokotna območja grafike z enega mesta na drugo, npr. ob vsaki spremembi položaja okna na zaslonu. Ta funkcija je zelo pomembna za udobno delo in jo večinoma že strojno podpirajo vse sodobnejše grafične kartice (*okenski pospeševalniki*). Kakorkoli že, ta nadvse uporabna in hitra funkcija je bila dolgo časa na voljo le kot zunanja funkcija API. V visual basicu 5 lahko za prepisovanje pravokotnih območij končno uporabimo tudi metodo **PaintPicture**.

Prenos vse slike iz gradnika v gradnik ni problematičen, saj lahko grafičnim vsebnikom in gradniku slika (*Image*) medsebojno prirejamo vsebino lastnosti Picture ali Icon. Zaplete se pri prenašanju izreza slike. Tedaj nam priskoči na pomoč metoda PaintPicture. Ne samo, da lahko prenese izrez, po potrebi ga lahko še raztegne ali skrči in prepiše na obstoječo podlago, upoštevajoč izbrani bitni operator. Popolna oblika metode je:

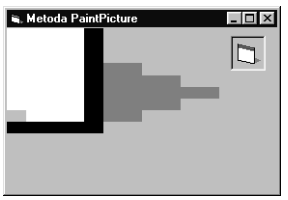
```
[predmet.]PaintPicture slika, _
  ciljX, ciljY, [ciljŠirina], [ciljVišina], _
  [izvorX], [izvorY], [izvorŠirina], [izvorVišina], _
  [bitnaOperacija]
```

Predmet mora biti grafični vsebnik ali predmet Printer, slika pa je lahko lastnost Picture ali Icon vseh gradnikov, ki jo poznajo. S prvimi štirimi koordinatami opredelimo ciljni položaj in velikost prenesene slike, druge štiri koordinate pa lahko določijo pravokotni izsek izvorne slike, ki jo želimo prepisati. Ker na metodo ne vpliva nastavitev lastnosti DrawMode, lahko bitno operacijo s podlago zapišemo eksplicitno.

Za preizkus metode na vmesnik namestimo gradnik Image (pustimo mu privzeto ime Image1) in prepišemo naslednjo kodo:

```
With Image1
  .Picture = Icon
  PaintPicture .Picture, 0, 0, ScaleWidth, ScaleHeight, _
   .Width \ 2, .Height \ 2
End With
```

Slika 10-12: Prenos izreza slike z vgrajeno metodo

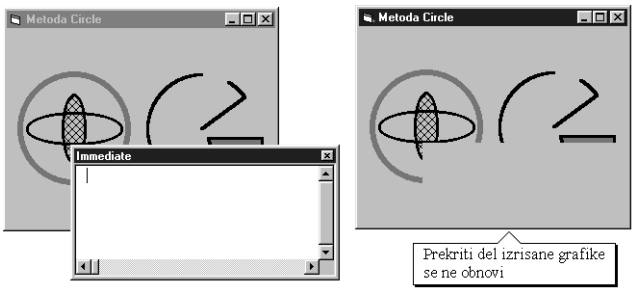


S to metodo smo prerisali spodnjo desno četrtino ikone čez vso površino obrazca. Zgled bi deloval, tudi če bi bil gradnik Image1 neviden.

Osveževanje grafike

Vsi dosedanji poizkusi dela z izrisano grafiko so imeli veliko pomanjkljivost: izrisana grafika ni bila *obstojna*. Obstojna (*persistent*) grafika sama skrbi za svoje obnavljanje, kadar je treba. Vsi gradniki znajo skrbeti sami zase, izrisana grafika pa ne. Vsaj ne takoj. Naredimo preprost preizkus. Enega izmed zgoraj zapisanih zgledov preizkusimo v bolj praktičnih delovnih razmerah:

Slika 10-13: Izrisana grafika ni obstojna.



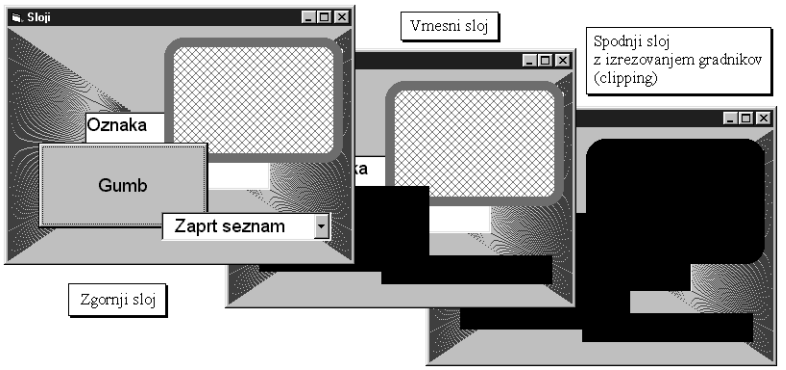
Če del okna prekrijemo ali vse okno pomanjšamo ter znova prikažemo, se izrisana grafika ne obnovi. S tem smo trčili ob bistveno težavo pri uporabi izrisane grafike. Rešimo jo lahko zelo preprosto, vendar si prej oglejmo sloje obrazca.

Sloji obrazca

Pri načrtovanju obrazcev lahko uporabljamo najrazličnejše gradnike, ki jih urejamo z ukazi menuja **Format ➪ Order**: **Bring To Front** in **Send To Back**. Tovrstno urejanje je mogoče tudi med izvajanjem programa, z metodo ZOrder. Kljub temu pri preurejanju gradnikov nimamo povsem prostih rok. Poizkusimo npr. postaviti oznako (*Label*) nad ukazni gumb (*CommandButton*). Ne bo nam uspelo. Vzrok je v slojih obrazca. Sloji so trije, od vrha proti dnu pa si sledijo takole:

1. Zgornji sloj. Gradniki, ki prejmejo vnosno točko.
2. Vmesni sloj. Grafični gradniki in oznaka.
3. Spodnji sloj. Grafika, izrisana z grafičnimi metodami med izvajanjem programa.

Slika 10-14: Ponazoritev slojev



Na vedenje posameznih slojev bistveno vplivata dve lastnosti, ki ju poznata grafična vsebnika obrazec in slikovni okvir (*PictureBox*) ter njun skupni dogodek Paint. Oglejmo si njihov kratek opis:

- Lastnost AutoRedraw. Ta lastnost določa, ali je izrisana grafika obstojna ali ne.
- Lastnost ClipControls. Ta lastnost določa, ali se pri izrisovanju grafike prepove uporaba območij, ki jih prekrivajo gradniki na zgornjih slojih, kar imenujemo *izrezovanje* (*clipping*).
- Dogodek Paint. Dogodek se pripeti vedno, kadar je treba osvežiti izrisano grafiko.

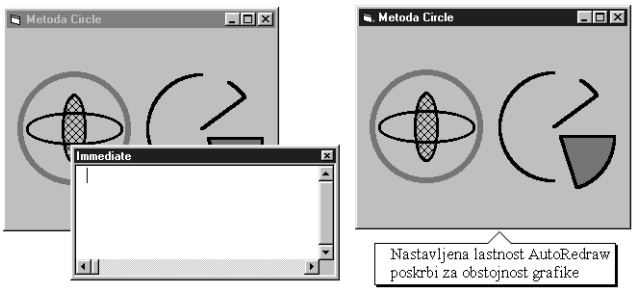
Skupni vpliv teh treh faktorjev povzema spodnja tabela:

Lastnost AutoRedraw	Lastnost ClipControls	Izrisovanje grafike samo v metodi Paint	Vpliv na sloje
True	True (privzeto)	Nepomembno	Običajno vedenje slojev.
True	False	Nepomembno	Običajno vedenje slojev. Obrazci, ki vsebujejo več neprekrivajočih se gradnikov, se bodo obnavljali celo nekoliko hitreje, ker je izrezovanje okrog gradnikov izključeno.
False (privzeto)	True (privzeto)	Da	Običajno vedenje.
		Ne	Grafični gradniki in učinki grafičnih metod se mešajo. Ni priporočljivo.
False	False	Da	Običajno vedenje. Obnavljajo se samo tisti deli obrazca, ki so bili prekriti ali so na novo prikazani (ob razširitvi okna).
		Ne	Vsi sloji se mešajo. Ni priporočljivo.

Učinki lastnosti AutoRedraw

Z nastavitvijo lastnosti AutoRedraw na resnično vrednost (*true*) se izognemo vsem opisanim težavam. Sloji se vedejo normalno, ne glede na mesto v kodi, kjer uporabljamo grafične metode. Izrisana grafika je obstojna.

Vendar za to udobje plačamo precejšen davek. Visual basic mora hraniti grafično vsebino obrazca v pomnilniku. Koliko pomnilnika je to, si bomo najlaže predstavljali, če izračunamo število zlogov, potrebnih za opis slike velikosti 640 × 480 pik pri visoki barvni ločljivosti (npr. 16-bitna barvna globina). Potrebujemo kar 600 KB. To je za današnje razmere razmeroma majhno okno. Uporaba nastavljene lastnosti AutoRedraw pri velikih obrazcih lahko povzroči občutne zamude zaradi velikih pomnilniških zahtev.



Slika 10-15: Obstojnost grafike lahko enostavno omogočimo z vklopom lastnosti AutoRedraw.

Učinki lastnosti ClipControls

Nastavljena lastnost ClipControls skrbi za to, da grafični gradniki in metode ne morejo vplivati na zgornji sloj. Pred vsakim izrisovanjem najde in prepove uporabo območij, ki jih prekrivajo gradniki na najvišjem sloju. Če je teh gradnikov precej, lahko izrisovanje znatno pospešimo z izklopom te lastnosti (vrednost *false*). Vendar moramo v tem primeru paziti, da se učinki grafičnih metod ne bodo mešali z gradniki na zgornjem sloju. To lahko rešimo na dva načina:

- Z nastavitvijo lastnosti AutoRedraw.
- Z omejitvijo uporabe grafičnih metod zgolj znotraj odziva na dogodek Paint.

Zadnja možnost je programersko resda malo zahtevnejša, vendar zagotavlja optimalne učinke, tako glede hitrosti kakor glede pomnilniške zahtevnosti. Če sta obe vplivni lastnosti izključeni (vrednost *false*), se obnavljajo le prej zakrita ali na novo prikazana območja obrazca.

> **NAMIG** Seveda je pri zahtevnih risbah obnavljanje slike znotraj odziva na dogodek Paint lahko dolgotrajno. Vse je prepuščeno spretnosti programerja. Tovrstno obnavljanje uporabljajo vsi programi za delo z vektorsko grafiko (AutoCAD, CorelDRAW...), uporabili pa ga bomo tudi v zgledu, ki ga bomo razvili v tem poglavju. Po drugi strani je za programe, ki obdelujejo bitne slike, ugodnejše pomnjenje celotne grafike v pomnilniku.

Vloga dogodka Paint

Če vse grafične metode uporabimo zgolj v odzivu na dogodek Paint, omogočimo visual basicu, da obnavlja obrazec po pravilnem vrstnem redu, kar odpravi vse nevšečnosti z mešanjem slojev, ne glede na nastavitev lastnosti AutoRedraw in ClipControls.

Uporabi grafičnih metod zunaj odziva na dogodek Paint se moramo izogibati. Izjema so posebni primeri, npr. obrazec, katerega vsebina se obnavlja po uri (gradnik *Timer*), različni ohranjevalniki zaslona, kratkotrajni grafični učinki in podobno. V vseh teh primerih je treba izrisano grafiko izolirati od drugih gradnikov. Najpogosteje je za to primeren poseben slikovni okvir (*PictureBox*).

Predmeta Screen in Printer

Vgrajena predmeta Screen in Printer omogočata dostop do dveh pomembnih sistemskih sredstev: zaslona in tiskalnika. Zaslon je v grafičnih uporabniških vmesnikih zelo občutljiv, zato imamo pri predmetu Screen, ki ga opisuje, na voljo le lastnosti, izmed katerih je večina na voljo zgolj za branje. Drugače je pri predmetu Printer. Kot smo lahko zaslutili že iz obravnave grafičnih metod, je predmet Printer zelo podoben grafičnima vsebnikoma (obrazec in slikovni okvir), saj nam operacijski sistem želi predstaviti vsako izhodno enoto zelo podobno. Nekatere posebnosti dela s tiskalnikom so omogočene z dodatnimi lastnostmi in metodami, ki jih bomo spoznali. Začnimo z zaslonom.

Zaslon

Zaslon opisuje predmet **Screen**, ki pozna le lastnosti. Te so:

Lastnost	Pomen
ActiveForm	Predstavlja trenutno dejavni obrazec programa.
ActiveControl	Predstavlja trenutno dejavni gradnik znotraj dejavnega obrazca programa.
Fonts	Polje z imeni vseh pisav, ki so na voljo za zaslon.
FontCount	Število vseh pisav, ki so na voljo za zaslon.
Height	Višina zaslona v twipih.
Width	Širina zaslona v twipih.
TwipsPerPixelX	Število twipov na eno zaslonsko piko vodoravno.
TwipsPerPixelY	Število twipov na eno zaslonsko piko navpično.
MousePointer	Kazalec miške.
MouseIcon	Uporabniško določen kazalec miške.

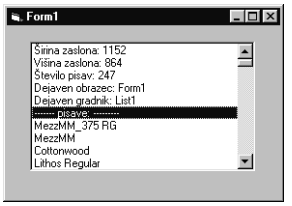
Opis nastavitve kazalca miške smo spoznali že v prvem delu knjige (obravnava istoimenskih lastnosti). Z nastavitvijo te lastnosti za predmet Screen spremenimo dejavni kazalec miške za ves program. Zato ga uporabljamo predvsem za označitev

posebnih stanj programa (npr. čakanje, ki ga ponazarja peščena ura, ki jo označuje vgrajena konstanta vbHourglass). Vsekakor moramo poskrbeti, da ga na koncu vrnemo v privzeto stanje (označuje ga konstanta vbDefault).

Podobno kakor smo pri gradniku MDIForm ugotavljali dejavni obrazec z lastnostjo ActiveForm, nam pri vmesnikih SDI pomaga istoimenska lastnost predmeta Screen. Pri obeh predmetih imamo na voljo tudi lastnost ActiveControl, ki še bolj natančno določi trenutni položaj vnosne točke (*focus*).

Zaradi združljivosti za nazaj so imena vseh pisav zbrana v polju nizov Fonts. Žal to ni prava zbirka, zato moramo sprehod po vseh pisavah zapisati z zanko For, število vseh pisav pa nam opredeli nova lastnost FontCount. Te in druge lastnosti bomo preizkusili v naslednjem zgledu. Za njegovo delovanje moramo na obrazec namestiti odprti seznam (pustimo mu privzeto ime List1):

```
Private Sub Form_Activate()
Dim i As Integer
  List1.Clear
  With Screen
    List1.AddItem "Širina zaslona: " & .Width \ .TwipsPerPixelX
    List1.AddItem "Višina zaslona: " & .Height \ .TwipsPerPixelY
    List1.AddItem "Število pisav: " & .FontCount
    List1.AddItem "Dejaven obrazec: " & .ActiveForm.Name
    List1.AddItem "Dejaven gradnik: " & .ActiveControl.Name
    List1.AddItem "------- pisave: --------- "
    For i = 0 To .FontCount - 1
      List1.AddItem Screen.Fonts(i)
    Next
  End With
End Sub
```



Slika 10-16: Uporaba lastnosti predmeta Screen

Tiskalniki

Tiskalnik je opisan s predmetom **Printer**. Ta predmet predstavlja privzeti tiskalnik. Vsi v sistemu dostopni tiskalniki so zbrani v zbirki (*collection*) **Printers**. Če želimo, lahko zbirko ročno pregledamo in poizkušamo najti najustreznejši tiskalnik, spoznali pa smo tudi standardna pogovorna okna, ki to nalogo prepustijo uporabniku.

Lastnosti

Predmet Printer pozna vrsto lastnosti, s katerimi lahko vplivamo na vse bistvene nastavitve tiskanja. Ker Okna in s tem visual basic obravnavajo tiskalnik precej abstraktno, je mogoče, da številne lastnosti ne bodo dale želenih učinkov. Vse je pač odvisno od tehničnih lastnosti tiskalnika in sposobnosti njegovega gonilnika (*device driver*). Kar nekaj izmed naštetih lastnosti v nadaljevanju velja preizkusiti na praktičnih zgledih.

Lastnost **DeviceName**

Opisno ime tiskalnika, ki ga predstavlja predmet Printer.

Lastnost **Port**

Z lastnostjo Port lahko preverimo zunanji priključek, s katerim je tiskalnik povezan z računalnikom.

Lastnosti **Fonts** in **FontCount**

Podobno kakor pri predmetu Screen ima tudi predmet Printer polje Fonts z imeni vseh pisav, ki so na voljo izbranemu tiskalniku. Skupno število pisav določa lastnost FontCount.

Lastnost **ColorMode**

Določa način izpisa barvnih tiskalnikov.

Lastnost **Copies**

S to lastnostjo lahko določimo število kopij izpisa.

Lastnost **Duplex**

Določi način izpisa na obe strani papirja, če to tiskalnik omogoča.

Lastnost **Orientation**

Lastnost določa pokončno ali ležečo orientacijo papirja.

Lastnost **Page**

Ta lastnost označuje zaporedno številko trenutno dejavne strani.

Lastnost **PaperBin**

Nastavitvi te lastnosti pripada 14 različnih vgrajenih konstant, ki opišejo želeni podajalnik papirja, ki bi ga pri izbranem tiskalniku radi uporabili za izpis.

Lastnosti **Width**, **Height** in **PaperSize**

S to lastnostjo in 41 vgrajenih konstant lahko izberemo značilne velikosti lista. Če nam nobena različica ne ustreza, lahko še vedno posežemo po lastnostih Width in Height. Učinek nastavitve je odvisen od tiskalnika in ga moramo preizkusiti.

Lastnost **PrintQuality**

Lastnost določa kakovost izpisa. Na voljo imamo štiri opisne vrednosti (predstavljene z vgrajenimi konstantami), po želji pa lahko kakovost določimo tudi neposredno, z nastavitvijo na želeno gostoto pik na palec (DPI, *dots per inch*). Učinek nastavitve je odvisen od tiskalnika oziroma njegovega gonilnika (*driver*), zato je potreben preizkus.

Lastnost **Zoom**

Lastnost določa odstotek povečave oziroma pomanjšave izpisa. Privzeta nastavitev je 0, kar pomeni običajno velikost. Nastavitev 50 bi izpis za polovico zmanjšala, 200 pa dvakrat povečala.

Lastnost **TrackDefault**

Če lastnost postavimo na resnično vrednost (*true*), se predmet Printer tudi med delom prilagaja privzetemu tiskalniku (če ga npr. med izvajanjem programa spremenimo s sistemskimi nastavitvami).

Vse omenjene vgrajene konstante z opisi najdete v sprotni pomoči.

Oglejmo si precej uporaben zgled:

```
Dim P As Object
For Each P In Printers
  If P.Port = "LPT2:" Or P.DeviceName Like "*LaserJet*" Then
    ' Nastavimo tiskalnik
    Set Printer = P
    Exit For
  End If
Next P
```

S sprehodom po zbirki najdemo prvi tiskalnik, ki je bodisi priključen na vrata LPT2 ali pa njegovo opisno ime vsebuje niz LaserJet. Najdeni tiskalnik izberemo v predmet Printer s stavkom Set. Odtlej je skozi predmet Printer dostopen natančno ta tiskalnik. Na splošno ni pomembno, ali lastnosti tiskalnika naslavljamo prek predmeta Printer ali prek posameznega elementa zbirke Printers. V zgornjem zgledu bi lahko tudi prepisali najdeno zaporedno številko v posebno globalno spremenljivko in jo uporabljali skozi poznejšo kodo za naslavljanje želenega tiskalnika v zbirki.

Metode

Predmet Printer predstavlja strojno neodvisno risalno površino znane velikosti, ki odgovarja trenutno izbrani velikosti papirja. Na njej lahko uporabimo vse grafične metode in lastnosti. Tiskanje je zato povsem podobno izrisovanju lastne grafike. Posebnost pri uporabi tiskalnika je le, da moramo po dokončno izrisani strani poklicati posebno metodo, ki določeno stran pošlje gonilniku, ta pa jo prevede v obliko, razumljivo tiskalniku.

Pomembne metode predmeta, ki jih še nismo natančneje opisali, so:

Metoda **NewPage**

Metoda NewPage konča trenutno stran, jo posreduje tiskalniku in poveča števec strani, shranjen v lastnosti Page.

Metoda **EndDoc**

Metoda EndDoc konča trenutno stran in ves izpis. Uporabimo jo na koncu izpisa. Brez škode (beri: dodatno izpisanega praznega lista) lahko uporabimo EndDoc tudi neposredno za klicem metode NewPage.

Metoda **KillDoc**

Z metodo KillDoc nemudoma prekličemo izpisovanje posredovanih strani. Če je v Oknih omogočeno tiskanje v ozadju (*background printing*) in se ukazi za tiskalnik še obdelujejo v upravniku tiskanja (*print manager*), bo metoda KillDoc povsem preprečila izpis. V drugih primerih je bil del ukazov že posredovan tiskalniku. Metoda KillDoc bo tedaj preprečila nadaljevanje tiskanja, kaj pa se bo zgodilo z že poslanimi ukazi, je odvisno od samega tiskalnika.

> **NAMIG** Zaradi sorodnosti med izpisovanjem grafike na zaslon in tiskalnik, lahko vsak izpis simuliramo na zaslonu. Ko smo zadovoljni z njegovim izidom, le zamenjamo predmet, ki je cilj izrisovanja, in ustrezno popravimo skalo. Z malo truda lahko enostavno napišemo funkcijo za vnaprejšnji ogled tiskanja (*print preview*).

Oglejmo si zgled, ki s tiskalnikom izpiše katalog razpoložljivih pisav, porazdeljen po straneh:

```
Private Sub IztiskajKatalog(strNiz As String)
Dim i As Integer
Dim intVrstic As Integer

  With Printer
    .Font.Size = 12
    ' Izračunamo število vrstic/stran
    intVrstic = .ScaleHeight \ .TextHeight(strNiz) \ 3
    For i = 0 To .FontCount - 1
      .Font.Name = .Fonts(1)
      .Font.Size = 12
      Printer.Print .Font.Name
      .Font.Name = .Fonts(i)
      .Font.Size = 24
      Printer.Print strNiz
      ' Lom strani?
      If (i + 1) Mod intVrstic = 0 Then
        Printer.Print
        Printer.Line -Step(ScaleWidth, 0)
        Printer.Print "Stran: " & .Page
```

```
        ' Konec strani
        .NewPage
      End If
    Next
    ' Konec tiskanja
    .EndDoc
  End With

End Sub
```

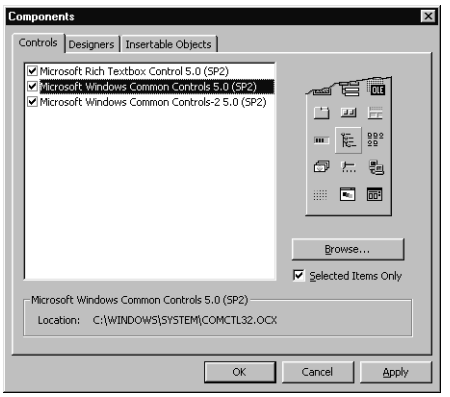
Tiskanje z visual basicom praktično nima omejitev. Edina težava je precejšnja količina kode, ki jo moramo zapisati za bolj oblikovane izpise.

Vmesnik za Okna 95

Okna 95 so prinesla nov in v marsičem boljši grafični uporabniški vmesnik. V visual basicu (velja za različici Professional in Enterprise) lahko z zunanjimi gradniki ActiveX pridobimo vse potrebne elemente, da bo naš program po videzu in funkcionalnosti povsem spominjal na druge programe, prilagojene Oknom 95.

Med nove gradnike za razvoj sodobnega vmesnika prištevamo razširitve iz treh zunanjih datotek (v projekt jih vključimo z menujskim ukazom **Project ➪ Components**...):

- Microsoft Windows Common Controls (gradniki ImageList, ListView, ProgressBar, Slider, StatusBar, TabStrip, ToolBar, TreeView).
- Microsoft Windows Common Controls-2 (gradniki Animation in UpDown).
- Microsoft Rich Textbox Control (gradnik RichTextBox).



Slika 10-17: Izbira zunanjih gradnikov za sodobnejši vmesnik

Da bi lahko uporabili zglede, ki sledijo, moramo odpreti nov projekt vrste standard EXE in mu ročno dodati zgoraj omenjene zunanje gradnike.

Drsnik in napredek

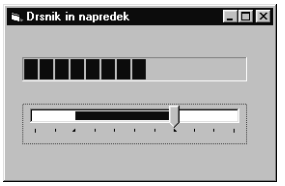
Najpreprostejša je uporaba gradnikov **ProgressBar** in **Slider**. Gradnik ProgressBar omogoča vizualno ponazoritev poteka časovno zahtevnejše operacije. Položaj znotraj intervala, ki ga omejujeta lastnosti Min in Max, določa lastnost Value. Za razliko od običajnih drsnikov ta lastnost pri ProgressBaru ni dostopna med načrtovanjem vmesnika.

Za izbiro iz intervala vrednosti lahko namesto običajnih drsnikov (*ScrollBars*) sedaj uporabimo tudi nov gradnik: Slider. Njegova prednost je v tem, da omogoča tudi izbor območja. Izbiranje območja je treba ročno sprogramirati. Nadzirajo ga lastnosti SelectRange, SelStart in SelLength, programiramo pa jih ponavadi tako, da jih nastavljamo s klikom ali vlečenjem jezička s hkrati pritisnjeno tipko Shift. Ustrezna dogodka sta MouseDown in MouseUp. Oglejmo si zgled (zgled predpostavlja, da sta na obrazcu izrisana gradnik ProgressBar in Slider s privzetima imenoma):

```
Private Sub Slider1_MouseDown(Button As Integer, Shift As Integer, _
x As Single, y As Single)
  With Slider1
    ' ob pritisnjeni tipki Shift
    If (Shift And vbShiftMask) Then
      ' omogočimo izbiranje območja
      .SelectRange = True
      ' brišemo morebitno staro območje
      .ClearSel
      ' nastavimo nov začetek območja
      .SelStart = .Value
    Else
      .ClearSel
    End If
  End With
End Sub

Private Sub Slider1_MouseUp(Button As Integer, Shift As Integer, x As Single, _
y As Single)
Dim intTemp
  With Slider1
    ' ob pritisnjeni dvigalki in izbranem označevanju območja
    If (Shift And vbShiftMask) And .SelectRange Then
    ' označimo lahko le pozitivne intervale
      If .Value >= .SelStart Then
        ' ob vlečenju desno je vse preprosto
        .SelLength = .Value - .SelStart
      Else
        ' pri vlečenju levo popravimo nastavitev začetka
        ' in uporabimo negativno negativno (=pozitivno) dolžino
        intTemp = .SelStart
        .SelStart = .Value
        .SelLength = -(.Value - intTemp)
      End If
    End If
  End With
End Sub
```

```
Private Sub Slider1_Change()
  ' ponazorimo območje v gradniku ProgressBar
  ProgressBar1.Value = Slider1.SelLength
End Sub
```

Slika 10-18: Vmesnik z gradnikoma za prikaz napredka in novim drsnikom, ki omogoča izbor podintervalov



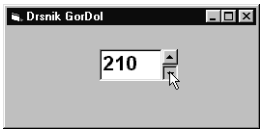
NAMIG Večina zunanjih gradnikov ActiveX, med katere sodita tudi obravnavana, ponudi v oknu za nastavljanje lastnosti še posebno nastavitev, imenovano (Custom). Ta nastavitev je pravzaprav pripomoček (imenovan *Property Pages*), ki prikaže posebno pogovorno okno, v katerem lahko pregledno nastavljamo več lastnosti, ki so smiselno razdeljene v skupine. Vse te lastnosti so na voljo tudi neposredno, vendar jih je s tem pripomočkom laže uporabljati.

Da, ta odstavek sem prepisal iz obravnave standardnih pogovornih oken. Če se je komu zdel znan, si zasluži pohvalo za zbrano branje.

Gor in dol po animacijah

V dodatni zunanji knjižnici, Microsoft Common Controls – 2, najdemo še dva gradnika, ki sta pogosta v sodobnih uporabniških vmesnikih. Prvi je zelo priročen gradnik **UpDown**, ki je v bistvu droben drsnik (vsebuje le dva gumba) z možnostjo neposredne navezave na poljubno lastnost poljubnega gradnika. Izberemo ga npr. kot spremljevalca vnosnega polja. To nakazuje naslednji zgled (zgled predpostavlja, da smo na vmesniku pripravili vnosno polje in gradnik UpDown ter jima pustili privzeta imena):

```
Private Sub Form_Load()
  With UpDown1
    .Min = 0
    .Max = 255
    .Increment = 10
    .BuddyControl = Text1.Name
    .BuddyProperty = "Text"
    .SyncBuddy = True
  End With
End Sub
```



Slika 10-19: Uporaba navezanega drsnika UpDown

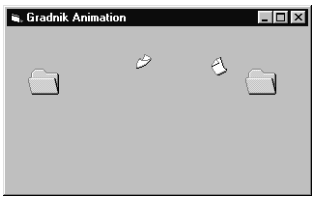
Gradnika sta sedaj med seboj povezana. Vrednosti v vnosnem polju je mogoče spreminjati tudi z uporabo gumbov gradnika UpDown1. Obseg intervala, ki ga pokriva, določimo z že znanima lastnostma Min in Max, spremembo pa z lastnostjo Increment. Če želimo, da se po intervalu vrednosti sprehajamo v krogu (minimumu sledi maksimum in nasprotno), nastavimo še lastnost Wrap. Ponavadi se izplača, da je sprememba večja vrednost (še posebej pri relativno velikih intervalih, kot je zgornji), saj lahko natančnejše vrednosti uporabnik vnese neposredno na seznam. Z nastavitvijo lastnosti SyncBuddy določimo, da se tudi neposredni vnosi v polje uskladijo s trenutno vrednostjo drsnika UpDown.

Vse lastnosti je mogoče še priročneje nastaviti v fazi načrtovanja vmesnika. Zelo uporabna lastnost je še AutoBuddy, ki samodejno izbere povezani gradnik in njegovo značilno lastnost na podlagi dostopnega zaporedja (*tab order*). Tudi učinek te lastnosti je neposredno opazen med načrtovanjem.

V zgoraj omenjeni knjižnici je še en gradnik: **Animation**. Ta nam omogoča, da popestrimo obrazce, podobno mnogim oknom novejših programov, npr. tistemu, ki spremlja kopiranje datotek v Raziskovalcu (*Explorer*). Kopiranje je ponazorjeno z animiranim preletom lista papirja. Tovrstne animacije morajo biti pripravljene v posebni obliki datoteke .AVI. Ta je lahko stisnjena kvečjemu po postopku RLE in *ne sme* vsebovati zvoka. Pri poizkusu predvajanja katerekoli druge podvrste zapisa .AVI bo gradnik sprožil izvajalno napako. Več pripravljenih animacij je priloženih visual basicu, najdemo jih v mapi Graphics\Avis\. Eno uporablja spodnji zgled (zgled predpostavlja, da je na vmesniku nameščen gradnik Animation1 in da smo projekt shranili v isto mapo, v katerem je tudi potrebna datoteka AVI):

```
Private Sub Form_Activate()
  Animation1.Open App.Path & "\filemove.avi"
End Sub

Private Sub Form_Click()
Static blnPredvaja
  If blnPredvaja Then
    Animation1.Stop
  Else
    Animation1.Play 3
  End If
  blnPredvaja = Not blnPredvaja
End Sub
```

Animacijo lahko odpremo le med izvajanjem programa z metodo Open. Predvajanje sprožimo z metodo Play, ki ji lahko posredujemo še število ponovitev ter začetno in končno sličico animacije. Če želimo, lahko nastavimo lastnost AutoPlay in



Slika 10-20: Predvajalnik animacij

predvajanje se bo začelo nemudoma po odpiranju datoteke. Na videz predvajanja vpliva še nekaj lastnosti, npr. Center in BackStyle. Pri načrtovanju obrazca moramo paziti, da velikost gradnika ustreza velikosti animacije.

Seznam slik

Neviden gradnik **ImageList** hrani *zbirko* (*collection*) predmetov razreda ListImage. Vsak izmed njih prestavlja eno sliko. Večina bolj zapletenih zunanjih gradnikov je programerju predstavljena kot zbirka predmetov. Gradnik redko uporabimo samostojno. Njegova najpogostejša vloga je shramba bitnih slik, ki jih pozneje uporabimo za prikaz vsebine v drugih gradnikih. Njegova resna pomanjkljivost je žal v tem, da morajo biti vse shranjene bitne slike enako velike. To velikost dokončno določi prva dodana slika.

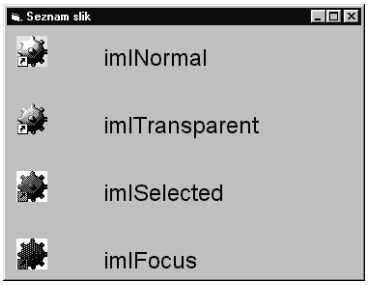
Bitne slike lahko gradniku dodamo že med oblikovanjem vmesnika – tu nam pomaga posebna nastavitev, združena lastnost (Custom) – ali pa za to poskrbimo v programski kodi. Gradnik ponuja metodo Draw, s katero lahko shranjeno bitno sliko prerišemo na drugo površino. To lahko storimo na več načinov:

- Neposredno. Izbrana slika se preriše nespremenjena.
- Prosojno. Barva, določena v lastnosti MaskColor, postane pri prerisovanju prosojna.
- Označeno (*selected*). Prerisana slika je osvetljena z barvo označevanja (določena je na ravni sistema), podobno, kot se označijo izbrane ikone na namizju ali v Raziskovalcu.
- Izbrano (*focused*). Prerisana slika v izbrani obliki je osvetljena in obrobljena s črtkano črto.

Različne načine prepisovanja uporablja naslednji zgled (zgled predpostavlja, da je na vmesniku izrisan prazen gradnik ImageList s privzetim imenom in da je projekt shranjen v isti mapi kakor slika v datoteki test.bmp):

```
Private Sub Form_Load()
Dim Image As ListImage
  ScaleMode = vbPixels
  Set Image = ImageList1.ListImages.Add(1, "slika", _
    LoadPicture(App.Path & "\test.bmp"))
```

```
  Font.Name = "Arial"
  Font.Size = 18
End Sub

Private Sub Form_Paint()
  CurrentX = 100
  CurrentY = 30
  Print "imlNormal"
  ImageList1.ListImages("slika").Draw hDC, 10, 10, imlNormal

  CurrentX = 100
  CurrentY = 100
  Print "imlTransparent"
  ImageList1.MaskColor = vbYellow
  ImageList1.ListImages("slika").Draw hDC, 10, 80, imlTransparent

  CurrentX = 100
  CurrentY = 170
  Print "imlSelected"
  ImageList1.ListImages("slika").Draw hDC, 10, 150, imlSelected

  CurrentX = 100
  CurrentY = 240
  Print "imlFocus"
  ImageList1.ListImages("slika").Draw hDC, 10, 220, imlFocus
End Sub
```



Slika 10-21: Demonstracija različnih načinov izrisovanja bitne slike z metodo Draw gradnika ImageList

V zgornjem zgledu smo se soočili z značilnim opravilom pri uporabi tovrstnih zunanjih gradnikov. Ker so pravzaprav zbirke predmetov, moramo napovedati začasno predmetno spremenljivko, potem pa ji prirejati izid funkcijske metode Add, ki v prvih dveh parametrih zahteva najprej enolično številčno, potem pa še tekstovno označitev elementa. Z njima lahko pozneje naslavljamo posamezen element zbirke.

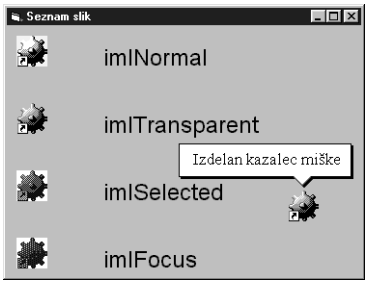
Metoda Draw zahteva ciljno risalno površino, opredeljeno s posebnim parametrom, *ročico risalne površine*. Ročica risalne površine (*handle to device context*, hDC) je na voljo pri vseh grafičnih vsebnikih, predmetu Printer in večini zunanjih gradnikov. V zgornjem zgledu smo uporabili kar ročico obrazca. Vse izrisovanje smo opravili znotraj metode Paint, zato sta lahko lastnosti AutoRedraw in ClipControls izključeni, slika pa se bo kljub temu pravilno sama obnavljala.

NAMIG Podobno kakor pri lastnosti hWnd tudi ročice hDC ne smemo prepisati v spremenljivko, ampak se moramo vedno sklicevati na lastnost.

Sliko, shranjeno v gradniku ImageList, lahko uporabimo tudi kot ikono. Metoda ExtractIcon postori vse delo. Njena uporaba je razvidna iz spodnjega zgleda, ki z novo ustvarjeno ikono nadomesti kazalec miške nad obrazcem:

```
Private Sub Form_Click()
' predmet, ki bo hranil sliko
Dim ico As Picture
  ' določimo prosojno barvo za kazalec
  ImageList1.MaskColor = vbYellow
  ' ico vsebuje sklic na novo ustvarjeni kazalec
  Set ico = ImageList1.ListImages(1).ExtractIcon
  ' priredimo ga obrazcu
  MousePointer = vbCustom
  MouseIcon = ico
End Sub
```

V zgornjem zgledu smo za začasno shrambo izdelane ikone uporabili poseben predmet **Picture**. Ta nam omogoča shraniti vse slike, ki jih lahko včitamo z metodo LoadImage, zgolj v pomnilniku, brez potrebe po dodatnem grafičnem gradniku ali vsebniku.



Slika 10-22: Rezultat metode ExtractIcon je bitna slika, ki jo lahko uporabimo za kazalec miške ali ikono.

Zanimiva metoda gradnika ImageList je še Overlay, ki naredi prekrivek dveh shranjenih slik, katerih oznaki posredujemo kot parametra. Pri prekrivanju uporabi prosojno barvo, kot jo določa lastnost MaskColor. Uporabo seznama slik bomo še podrobneje spoznali pri zgledu, ki nas čaka v finalu tega poglavja.

Večlična okna, drevesa in seznami

Sedaj si oglejmo še uporabo gradnikov TabStrip, ListView in TreeView. Pri vseh si lahko vnaprej pripravimo gradnik ImageList, ki ga zapolnimo s slikami, bodisi med načrtovanjem vmesnika ali programsko.

Gradnik **TabStrip** omogoča izdelovanje večstranskih pogovornih oken, podobnih nastavitvam projekta v visual basicu. Tako kakor je na seznamu slik vsaka posamezna slika predmet, gradnik pa je v bistvu njihova zbirka, je tudi pri gradniku TabStrip vsak jeziček predmet razreda Tab. Vsak jeziček ima lahko napis in bitno sliko, če ga povežemo s seznamom slik. Tega določa lastnost ImageList in mora vsebovati ime obstoječega gradnika ImageList. Po tem za vsak jeziček določimo pripadajočo sliko (lastnost Image).

Pri uporabi gradnika TabStrip se moramo zavedati, da gradnik sam po sebi ni vsebnik (*container*) in zato ne zna preklapljati s strani na stran. Skrbi samo za izris pravega gumba, vsebino pa moramo prikazovati s programsko kodo. Za vsebnike uporabimo ponavadi polje gradnikov PictureBox, narisanih drugo nad drugim, ki vsak zase vsebuje eno stran obrazca. Preklop strani servisiramo v odzivni proceduri klika gradnika TabStrip. Največkrat je ta procedura videti takole:

```
Private Sub TabStrip1_Click()
  Picture1(TabStrip1.SelectedItem.Index - 1).ZOrder 0
End Sub
```

Pri kliku postavimo na vrh (z metodo ZOrder) gradnik, ki po indeksu ustreza jezičku gradnika TabStrip. Naš zgled predpostavlja, da ima polje gradnikov PictureBox izhodiščno zaporedno številko 0, medtem ko so jezički vedno označeni od zaporedne številke 1 naprej.

> **NAMIG** Uporaba gradnika TabStrip zna biti zares zapletena. Pravzaprav je njegova pasivnost naravnost nedopustna, saj so vsi zunanji gradniki ActiveX lahko »živi« že med načrtovanjem programa. Očitno se je te slabosti zavedel tudi Microsoft, zato nam je priložil še poseben zunanji gradnik, ki ga skriva datoteka Microsoft Tabbed Dialog Control. Edini gradnik v njej – gradnik SSTab – je mnogo bolj uporaben, saj mu strani lahko popolnoma določimo že med načrtovanjem vmesnika.

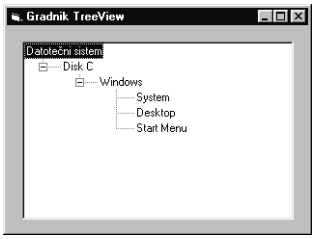
Natančneje si oglejmo uporabo dveh seznamov, ki sta hkrati tudi sestavni del vmesnika po zgledu Raziskovalca (*Explorer*). Levi del, ki prikazuje drevesno strukturo, je v visual basicu predstavljen z gradnikom TreeView. Desni del, seznam elementov v štirih različnih oblikah, pa je omogočen z gradnikom ListView.

Drevesna struktura gradnika **TreeView** je določena z zbirko predmetov razreda Node. Oglejmo si zgled programske kode, ki vzpostavi začetni videz drevesa:

```
Private Sub Form_Load()
' predmet za dodajanje vozlišč drevesa
Dim nodX As Node
  ' izhodišče drevesa, podamo ime in sliko
  Set nodX = TreeView1.Nodes.Add(, , "root", "Datotečni sistem")
  ' dodamo podrejeni element
  Set nodX = TreeView1.Nodes.Add("root", tvwChild, "c", "Disk C")
```

```
  ' naslednji element je podrejen elementu z oznako "c"
  Set nodX = TreeView1.Nodes.Add("c", tvwChild, "win", "Windows")
  ' preostali elementi so podrejeni "win"
  Set nodX = TreeView1.Nodes.Add("win", tvwChild, , "System")
  ' zahtevamo, da je prej dodani element viden, zato bo vidna
  ' celotna podveja, ki ji pripada
  nodX.Ensurevisible
  Set nodX = TreeView1.Nodes.Add("win", tvwChild, , "Desktop")
  Set nodX = TreeView1.Nodes.Add("win", tvwChild, , "Start Menu")
End Sub
```

Slika 10-23: Videz drevesnega prikaza ob zagonu programa



Vsako vozlišče je predmet tipa Node, zbrani pa so v zbirki Nodes. Po njej se najlaže sprehodimo z zanko For Each. Poljubno vozlišče lahko prisilimo v vidno stanje z metodo EnsureVisible. Stanje vozlišča (odprto ali zaprto) vrne lastnost Expanded. Nadrejenemu vozlišču lahko z nastavitvijo lastnosti Sorted nastavimo, naj uredi vrstni red svojih potomcev. Naslednji zgled uredi celotno drevo:

```
Dim n As Node
For Each n In TreeView1.Nodes
  If n.Children > 1 Then n.Sorted = True
Next
```

Kjerkoli je več potomcev, vklopimo urejenost. Drugo samodejno postori gradnik.

Če gradnik TreeView ponuja podobno funkcionalnost kakor leva stran Raziskovalca (drevo datotečnega sistema), je na desni strani podoben gradnik ListView (seznam datotek). Gradnik **ListView** omogoča pregledovanje poljubnega seznama v štirih različnih pogledih:

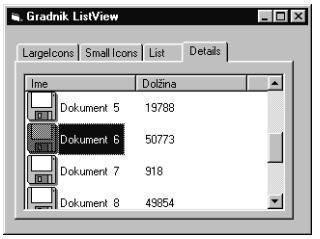
- Velike ikone in opis.
- Manjše ikone in opis.
- Seznam opisov.
- Poročilo, ki vključuje opis in dodatne podatke (npr. dolžina in datum datotek).

Vse različne velike in manjše ikone shranimo v gradnik ImageList, ki ga povežemo z ustreznim gradnikom ListView. Način predstavitve seznama določa lastnost View. Vsak element seznama je predmet ListItem, poleg teh pa lahko v gradnik ListView vključimo še predmete ColumnHeader, ki pojasnjujejo vsebino (vidni bodo le v pogledu poročila).

Seznam lahko zapolnimo med oblikovanjem vmesnika – z uporabo lastne nastavitve (Custom), gre pa tudi med izvajanjem programa s programsko kodo (naslednji zgled predvideva, da so na vmesniku gradniki TabList, ListView in ImageList, vsi s privzetimi imeni, v gradnik ImageList1 pa je že včitana ena slika):

```
Private Sub Form_Load()
Dim clmX As ColumnHeader
Dim itmX As ListItem
Dim i As Integer

  ' določimo pojasnili
  Set clmX = ListView1.ColumnHeaders.Add()
  clmX.Text = "Ime"
  Set clmX = ListView1.ColumnHeaders.Add()
  clmX.Text = "Dolžina"

  ' zapolnimo seznam
  For i = 1 To 10
    Set itmX = ListView1.ListItems.Add()
     itmX.SmallIcon = 1
     itmX.Icon = 1
    ' opis je v lastnosti Text in je viden vedno
    itmX.Text = "Dokument " & Str(i)
    ' podelementi so vidni le v izpisu v obliki poročila
    ' zapolnimo jih z naključnimi "dolžinami"
    itmX.SubItems(1) = Str(Int(Rnd * 65536))
  Next i

End Sub

' Preklop različnih pogledov smo omogočili kar
' prek gradnika TabStrip
Private Sub TabStrip1_Click()
  ListView1.View = TabStrip1.SelectedItem.Index - 1
End Sub
```

Slika 10-24: Zgled rabe seznama z različnimi pogledi na vsebino

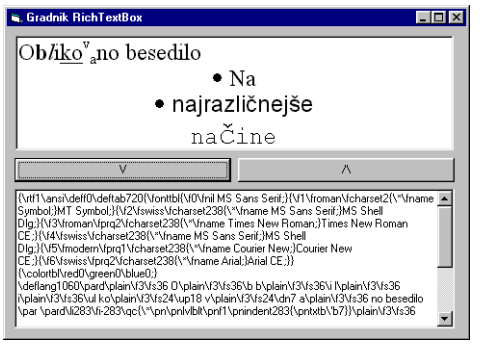


Oblikovano besedilo

Na tem mestu si oglejmo še gradnik **RichTextBox**. Najdete ga v ločeni knjižnici Microsoft Rich Textbox Control. Gradnik je v bistvu podoben običajnemu vnosnemu polju (*TextBox*), le da lahko prikazuje tudi oblikovano besedilo. Tega nastavljamo ali preberemo prek lastnosti TextRTF, zapišemo pa ga v posebni obliki, imenovani *rich text*. Ta format je v rabi v večini Microsoftovih izdelkov, dobro pa ga

poznamo vsi pisci datotek s sprotno pomočjo. Ročno kodiranje v RTF je zelo utrudljivo in ga nikomur ne priporočam. Na spodnji sliki vidite zgled izvornega in upodobljenega besedila.

Slika 10-25: Zgled rabe gradnika RichTextBox



Na srečo se lahko ročnemu kodiranju udobno izognemo. Gradnik v celoti podpira operacije »povleci in izpusti«, tako da lahko oblikovano besedilo iz npr. Worda z miško označimo, povlečemo in izpustimo nad gradnikom (ta operacija ne deluje med načrtovanjem vmesnika, program morate najprej zagnati). Izvorno besedilo je sedaj vsebovano v lastnosti TextRTF in ga lahko preprosto prepišemo v drug gradnik, na odložišče, ali pa kar v datoteko:

```
Open "Besedilo.RTF" For Output As #1
Print #1, RichTextBox1.TextRTF
Close #1
```

Sedaj ga lahko preučimo in po potrebi popravimo.

Gradnik RichTextBox ima ustrezne lastnosti in metode, s katerimi lahko nadziramo oblikovanje besedila. Med njimi najdemo nastavitev poravnave za vsak odstavek (leva, desna, sredinska), izbiro atributov znakov (krepko, ležeče, podčrtano, prečrtano), dodajanje alinej (*bullets*) in spreminjanje barve besedila. Natančnejšo dokumentacijo najdemo v sprotni pomoči. Tako kakor smo z vnosnim poljem razvili program Urejevalnik, izboljšano različico okenske beležke, bi z gradnikom RichTextBox lahko razvili popoln grafičen urejevalnik besedila, podoben okenskemu urejevalniku WordPad.

NAMIG Če smo že ob opisu zmožnosti dodatnih zunanjih gradnikov zaslutili, da nam brez njih ni več živeti in nas moti njihovo vklapljanje ob začetku vsakega novega projekta vrste standard EXE, lahko težavico zelo hitro odpravimo. Lotimo se novega projekta, vklopimo želene zunanje gradnike, zatem pa *projekt* shranimo kot predlogo. Ustrezna mapa je Template\Projects\.

Zglede kode, uporabljene pri opisu novih zunanjih gradnikov, najdete na priloženi plošči CD-ROM v mapi \Primeri\2-Delavnica\3-Risar\.

Zgled Risar

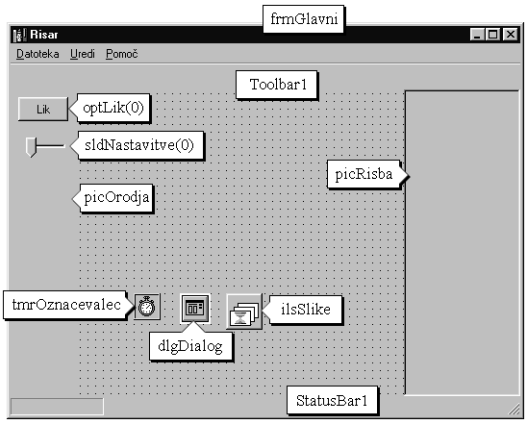
Po tem precej utrudljivem uvodu se bomo spopadli s pripravo uporabniškega vmesnika, definicijo podatkovnih struktur in pisanjem kode za Risarja, program za risanje slik, ki povzema značilnosti vmesnika programov v Oknih 95. Razviti program bo imel kar nekaj posebnih zmožnosti. Za celotno obnavljanje izrisane slike bo skrbel sam, omogočal bo razveljavljanje in vnovično uveljavljanje ukazov brez omejitev, celo pri že shranjeni datoteki. Uporabljal bo čisto svoj zapis vsebine v datoteko s podaljškom .RIS. Omogočal bo prenos slike prek odložišča in izvoz v obliki bitne slike. To pa še ni vse. Risar bo izdatno uporabljal datoteko sredstev, ki omogoča enostavnejšo izvedbo in poznejšo prilagoditev programa, na koncu pa mu bomo dodali še čisto pravo sprotno elektronsko pomoč in celoten program pripravili za razpečevanje.

Uporabniški vmesnik

Odprimo nov projekt vrste standard EXE. Za izdelavo Risarja bomo potrebovali dodatne gradnike iz dveh zunanjih knjižnic:

- Microsoft Common Dialog Control (standardna pogovorna okna),
- Microsoft Windows Common Controls (gradniki za sodoben vmesnik).

Ko jih vključimo v projekt (**Project ➪ Components**...), se lahko lotimo priprave uporabniškega vmesnika. Oblikujmo ga po vzoru s spodnje slike:



Slika 10-26: Začetni uporabniški vmesnik

Če se ozremo na sliko, bomo opazili, da bo njegov dokončen videz določila programska koda. Že sedaj pa je razvidno, da se bo risanje odvijalo v posebnem vsebniku, slikovnem gradniku picRisba. Uporabili bomo še en slikovni okvir, picOrodja, ki bo rabil kot vsebnik orodjarne. Oglejmo si uporabljene gradnike in nastavitve njihovih lastnosti:

Gradnik	Lastnost	Nastavitev
obrazec (*Form*)	(Name)	frmGlavni
	Caption	"Risar"
	Icon	(datoteka frmGlavni.ico)
dialogi (*CommonDialog*)	(Name)	dlgDialog
seznam slik (*ImageList*)	(Name)	ilsSlike
ura (*Timer*)	(Name)	tmrOznacevalec
	Enabled	False
	Interval	100
orodjarna (*Toolbar*)	(Name)	Toolbar1
	Align	1 - Align Top
	Height	420
	Appearance	1 - cc3D
vrstica stanja (*StatusBar*)	(Name)	StatusBar1
	Align	2 - Align Bottom
	Height	300
slikovni okvir (*PictureBox*)	(Name)	picRisba
	Align	4 - Align Right
slikovni okvir (*PictureBox*)	(Name)	picOrodja
	Align	3 - Align Left
	BorderStyle	0
	Width	1000
drsnik 95 (*Slider*)	(vsebovan v picOrodja)	
	(Name)	sldNastavitve
		Index 0
opcijsko stikalo (*OptionButton*)	(vsebovan v picOrodja)	
	(Name)	optLik
	Index	0
	Style	1 - Graphical

Risar bo uporabljal dve različno izdelani orodjarni. Zgornja vrstica gumbov bo uporabljala zunanji gradnik Toolbar, stranska orodjarna pa bo izdelana z običajnimi vgrajenimi izbirnimi stikali, le da bodo izrisana v grafični obliki (glej nastavitev lastnosti Style gradnika optLik).

Pri načrtovanju Risarja se bomo skušali kar najbolj opreti na nastavljanje lastnosti iz programske kode. Število vseh lastnosti, ki jih je treba nastaviti, je enostavno preveliko za pregledno obravnavo med načrtovanjem vmesnika. Vzpostavljanje gradnikov bomo zato prepustili podprogramom. Pri uporabi gradnikov za sodobnejši uporabniški vmesnik, ki smo jih spoznali malo prej, smo že lahko opazili, da je treba kar nekaj vrstic kode za določitev najosnovnejše funkcionalnosti gradnika. Tu uporabljamo štiri tovrstne gradnike: novi drsnik, seznam slik ter vrstici stanja in gumbov.

Ne pozabimo še na glavni menu. Njegovi ročni izdelavi se le stežka izognemo.

Glavni menu

V spodnji tabeli je povzetek definiranih menujev za glavno okno:

Ime menuja	Ime izbire[(zap. št.)]	Napis	Bližnjica
mnuDatoteka		"&Datoteka"	
	mnuDatotekaNova	"&Nova"	Ctrl+N
	mnuDatotekaOdpri	"&Odpri..."	Ctrl+O
	mnuDatotekaShrani	"&Shrani"	Ctrl+S
	mnuDatotekaShraniKot	"Shrani &kot..."	
	mnuDatotekaIzvozi	"&Izvozi"	
	mnuDatotekaIzvoziKotBitnoSliko	"Kot &bitno sliko..."	
	mnuLocilo1	"-"	
	mnuDatotekaKonec	"&Konec"	
mnuUredi		"&Uredi"	
	mnuUrediRazveljavi	"&Razveljavi"	Del
	mnuUrediUveljavi	"&Uveljavi"	Shift+Del
	mnuLocilo2	"-"	
	mnuUrediKopiraj	"&Kopiraj"	Ctrl+C
mnuPomoc		"&Pomoč"	
	mnuPomocKazalo	"&Kazalo..."	
	mnuPomocPojasnila "&Pojasnila"		
	mnuLocilo3	"-"	
	mnuPomocvizitka	"&Vizitka..."	

Ime menuja	Ime izbire[(zap. št.)]	Napis	Dodatno
mnuPrirocni		"Priročni meni"	Visible = False
	mnuPrirocniUkaz(0)	"&Označevanje"	
	mnuPrirocniUkaz(1)	"&Prostoročno risanje"	
	mnuPrirocniUkaz(2)	"&Ravne črte"	
	mnuPrirocniUkaz(3)	"Pravo&kotniki"	
	mnuPrirocniUkaz(4)	"&Elipse"	
	mnuLocilo4	"-"	
	mnuPrirocniBarve	"&Barve"	
	mnuPrirocniBarveIzbira(0)	"&Rdeča"	
	mnuPrirocniBarveIzbira(1)	"&Rumena"	
	mnuPrirocniBarveIzbira(2)	"&Zelena"	
	mnuPrirocniBarveIzbira(3)	"&Modra"	
	mnuLocilo5	"-"	
	mnuPrirocniDebeline	"&Debeline"	
	mnuPrirocniDebelineIzbor(0)	"&Lasnica"	
	mnuPrirocniDebelineIzbor(1)	"&Tanko"	
	mnuPrirocniDebelineIzbor(2)	"&Debelejše"	
	mnuPrirocniDebelineIzbor(3)	"&Najdebelejše"	

Poleg običajnih elementov glavnega menuja smo določili še *nevidni* menu mnuPrirocni, ki ga bomo uporabili za prikaz priročnega menuja nad risalno površino.

> **NAMIG** Izvorno kodo izdelanega vmesnika najdete na priloženi plošči CD-ROM v mapi \Primeri\2-Delavnica\3-Risar\1-Vmesnik\.

Vzpostavitve ob zagonu

Prva naloga programske kode bo dokončna vzpostavitev vseh gradnikov. Zahtevana opravila smo razdelili na več pomožnih podprogramov, ki jih kličemo iz dogodka Load:

```
' Začetne nastavitve ob nalaganju obrazca
Private Sub Form_Load()

  VzpostaviSlikanico
  VzpostaviGumbnico
  VzpostaviOrodjarno
  VzpostaviVrsticoStanja
```

```
  With dlgDialog
    .Filter = "Risarije (*.ris)|*.ris|Vse datoteke (*.*)|*.*"
    .DefaultExt = ".ris"
  End With

  ' Zaključne vzpostavitve
  Caption = App.Title
  ScaleMode = vbPixels
  Move 0.05 * Screen.Width, 0.05 * Screen.Height, _
    0.8 * Screen.Width, 0.8 * Screen.Height

End Sub
```

Pokličemo štiri vzpostavitvene podprograme, zatem pa uredimo še nastavitve standardnih pogovornih oken ter napis, skalo in začetni položaj glavnega okna.

Prvi vzpostavitveni podprogram napolni seznam slik:

```
' Vzpostavitev gradnika s seznamom slik
Sub VzpostaviSlikanico()
Dim imgTmp As ListImage
  ' Slike bomo prebrali v gradnik iz datoteke sredstev
  With ilsSlike.ListImages
    Set imgTmp = .Add(, , LoadResPicture(1001, vbResBitmap))
    Set imgTmp = .Add(, , LoadResPicture(1002, vbResBitmap))
    Set imgTmp = .Add(, , LoadResPicture(1003, vbResBitmap))
    Set imgTmp = .Add(, , LoadResPicture(1004, vbResBitmap))
    Set imgTmp = .Add(, , LoadResPicture(1005, vbResBitmap))
    Set imgTmp = .Add(, , LoadResPicture(1006, vbResBitmap))
    Set imgTmp = .Add(, , LoadResPicture(1007, vbResBitmap))
    Set imgTmp = .Add(, , LoadResPicture(1008, vbResBitmap))
  End With
End Sub
```

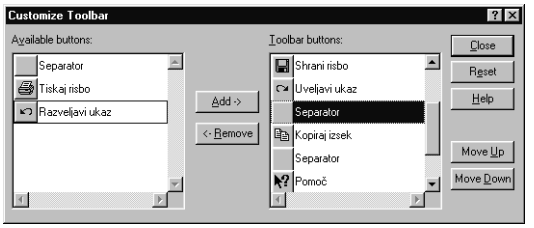
Videti je znano. Znova moramo ustvarjati več izvodov predmeta ListImage, ki ga uvrščamo v zbirko, določeno v gradniku ListImage, z metodo Add. Novost je funkcija **LoadResPicture**. Z njo preberemo sliko iz datoteke sredstev (*resource file*). Preden se poglobimo v pripravo in uporabo datoteke sredstev, si oglejmo še druge vzpostavitve. Naslednja je na vrsti »gumbnica«:

```
' Vzpostavitev vrstice z gumbi
Sub VzpostaviGumbnico()
Dim btnTmp As Button

  ' Določimo vir slik
  Toolbar1.ImageList = ilsSlike

  ' Dodamo gumbe
  With Toolbar1.Buttons
    Set btnTmp = .Add(, , , tbrSeparator)
    Set btnTmp = .Add(, "Nova", , tbrDefault, 1)
    btnTmp.Description = "Nova risba"
    Set btnTmp = .Add(, "Odpri", , tbrDefault, 2)
    btnTmp.Description = "Odpri risbo"
    Set btnTmp = .Add(, "Shrani", , tbrDefault, 3)
    btnTmp.Description = "Shrani risbo"
```

```
    Set btnTmp = .Add(, "Tiskaj", , tbrDefault, 4)
    btnTmp.Description = "Tiskaj risbo"
    Set btnTmp = .Add(, , , tbrSeparator)
    Set btnTmp = .Add(, "Razveljavi", , tbrDefault, 5)
    btnTmp.Description = "Razveljavi ukaz"
    Set btnTmp = .Add(, "Uveljavi", , tbrDefault, 6)
    btnTmp.Description = "Uveljavi ukaz"
    Set btnTmp = .Add(, "Kopiraj", , tbrDefault, 7)
    btnTmp.Description = "Kopiraj izsek"
    Set btnTmp = .Add(, , , tbrSeparator)
    Set btnTmp = .Add(, "Pomoc", , tbrDefault, 8)
    btnTmp.Description = "Pomoč"
  End With

  ' Opišemo gumbe s pojasnili
  With Toolbar1
    ' "Gumb" 1 je ločilo
    .Buttons(2).ToolTipText = "Nova risba"
    .Buttons(3).ToolTipText = "Odpri risbo"
    .Buttons(4).ToolTipText = "Shrani risbo"
    .Buttons(5).ToolTipText = "Natisni risbo"
    ' "Gumb" 5 je ločilo
    .Buttons(7).ToolTipText = "Razveljavi"
    .Buttons(8).ToolTipText = "Uveljavi"
    .Buttons(9).ToolTipText = "Kopiraj"
    ' "Gumb" 10 je ločilo
    .Buttons(11).ToolTipText = "Pomoč"
  End With

End Sub
```

Dodajanje gumbov v gradnik Toolbar je trifazno opravilo. Najprej moramo ustvariti nov predmet Button, ki sodi v zbirko. Če je ta gumb zgolj ločilo (tbrSeparator), smo tudi končali. Pri pravih gumbih se izplača nastaviti še dve lastnosti: opis (Description) in namig (ToolTipText). Gradnik Toolbar zna prikazovati namige, če jih ustrezno definiramo. Zna pa še nekaj: uporabnik ga lahko po želji preureja. Če bomo preurejanje dopustili, moramo vsak dodani gumb opisati še z nastavitvijo lastnosti Description.

Dopuščanje preurejanja orodjarne določa lastnost gradnika Toolbar, imenovana **AllowCustomize**. Če je nastavljena na resnično vrednost, bo uporabnik med izvajanjem programa z dvakratnim klikom orodjarne lahko odprl okno za njeno preurejanje. Žal je to okno izdelano v tuji jezikovni različici (visual basica zaenkrat še niso prevedli v slovenščino):



Slika 10-27: Okno za prirejanje vrstice z gumbi. Vidni so določeni opisi gumbov.

To ni edini način preurejanja. Uporabnik lahko gumbe prestavlja tudi z miško, če na tipkovnici hkrati tišči tipko Shift. Kakorkoli že, če nam tovrstno preurejanje ni všeč, ga izklopimo z nastavitvijo AllowCustomize.

> **NAMIG** Čeprav se zdi možnost preurejanja orodjarn na prvi pogled privlačna, ni preveč uporabna, saj jo poleg jezikovne neusklajenosti pesti še hujša težava: preureditve orodjarne se ne zapišejo, tako da nas ob vnovičnem zagonu programa znova pričaka privzeta nastavitev. Če je že shranjevanje sprememb ostalo na naših ramenih, lahko tudi prilagajanje izvedemo po svoje in se izognemo vgrajenemu preurejevalniku.
>
> Čisto zasebno – uporaba gradnika Toolbar se mi zdi nasploh odveč. Raje uporabimo grafično obliko vgrajenih gumbov in stikal, tako kakor bomo storili pri stranski orodjarni.

Naslednji element uporabniškega vmesnika, ki čaka na vzpostavitev, je stranska orodjarna, vsebovana v slikovnem okviru. Namesto zunanjega gradnika ActiveX smo za gumbe uporabili kar vgrajeni gradnik – opcijsko stikalo v grafični obliki. Videz orodjarne smo »začinili« še s skupino novih drsnikov. Oglejmo si izpis podprograma:

```
' Vzpostavitev orodjarne
Sub VzpostaviOrodjarno()
  ' Za lažjo razmestitev uporabimo točke
  picOrodja.ScaleMode = vbPixels

  ' Dodamo posamezna orodja
  ' - prvo zahteva zgolj nastavitev
  optLik(0).Move 8, 8, 50, 36
  optLik(0).Caption = ""
  optLik(0).ToolTipText = "Označevanje"
  optLik(0).Picture = LoadResPicture(1001, vbResCursor)

  ' - druga dodamo dinamično
  Load optLik(1)
  optLik(1).Top = optLik(0).Top + 36
  optLik(1).Caption = ""
  optLik(1).ToolTipText = "Prostoročno risanje"
  optLik(1).Picture = LoadResPicture(1002, vbResCursor)
  optLik(1).Visible = True

  Load optLik(2)
  optLik(2).Top = optLik(1).Top + 36
  optLik(2).ToolTipText = "Ravne črte"
  optLik(2).Picture = LoadResPicture(1003, vbResCursor)
  optLik(2).Visible = True

  Load optLik(3)
  optLik(3).Top = optLik(2).Top + 36
  optLik(3).ToolTipText = "Pravokotniki"
```

```
  optLik(3).Picture = LoadResPicture(1004, vbResCursor)
  optLik(3).Visible = True

  Load optLik(4)
  optLik(4).Top = optLik(3).Top + 36
  optLik(4).ToolTipText = "Elipse"
  optLik(4).Picture = LoadResPicture(1005, vbResCursor)
  optLik(4).Visible = True

  ' Drsniki nastavitev
  ' - tudi tu je prvi pripravljen
  With sldNastavitve(0)
    .Min = 0
    .Max = 255
    .SmallChange = 1
    .LargeChange = 20
    .Move 2, optLik(4).Top + 50, 62, 20
    .ToolTipText = "Delež rdeče barve"
  End With

  ' - druge naložimo dinamično
  Load sldNastavitve(1)
  With sldNastavitve(1)
    .Top = sldNastavitve(0).Top + 20
    .ToolTipText = "Delež zelene barve"
    .Visible = True
  End With

  Load sldNastavitve(2)
  With sldNastavitve(2)
    .Top = sldNastavitve(1).Top + 20
    .ToolTipText = "Delež modre barve"
    .Visible = True
  End With

  ' - zadnji je za debelino črte
  Load sldNastavitve(3)
  With sldNastavitve(3)
    ' - ni povsem po vzoru prvega...
    .Min = 1
    .Max = 20
    .SmallChange = 1
    .LargeChange = 5
    .Top = sldNastavitve(2).Top + 50
    .ToolTipText = "Debelina črte"
    .Visible = True
  End With

End Sub
```

Izbiranje risalnih orodij omogočimo s skupino izbirnih stikal v grafični obliki. Njihovo podobo določa bitna slika, prebrana iz že omenjene zunanje datoteke sredstev. Na dnu dodamo še dve skupini drsnikov. V prvi so trije za izbiro barvnega odtenka, vsak omogoča izbiranje v intervalu 0–255. Zadnji drsnik je nameščen nekoliko niže in omogoča izbiranje v ožjem intervalu, ker rabi za določitev debeline pisala. Kakor vidimo, je določanje vseh bistvenih lastnosti gradnika, od položaja do

videza, izdelano v programski kodi. Videz vmesnika med načrtovanjem tako sploh ni bistven, pomembno je le, da imamo pripravljen prvi izvod bodočega polja gradnikov, da lahko med izvajanjem programa dinamično dodajamo nove elemente, jih preoblikujemo in prikažemo.

Zadnji izmed bolj zapletenih gradnikov, potrebnih vzpostavitve, je še vrstica stanja, gradnik StatusBar:

```
' Vzpostavitev vrstice stanja
Sub VzpostaviVrsticoStanja()
Dim pnlTmp As Panel

  ' Dodamo elemente vrstice stanja
  With StatusBar1.Panels
    Set pnlTmp = .Add(, "ura", "", sbrTime)
    Set pnlTmp = .Add(, "datum", "", sbrDate)
  End With

  ' Nastavitve posameznih elementov
  With StatusBar1
    .Panels(1).MinWidth = 700
    ' Poravnava na levo stran
    .Panels(1).Alignment = sbrLeft
    ' Razprostri se čez razpoložljivo širino
    .Panels(1).AutoSize = sbrSpring

    .Panels(2).Bevel = sbrNoBevel
    ' Sredinska vodoravna poravnava
    .Panels(2).Alignment = sbrCenter
    ' Ohranjaj širino, potrebno za prikaz vsebine
    .Panels(2).AutoSize = sbrContents

    .Panels(3).Bevel = sbrNoBevel
    .Panels(3).Alignment = sbrCenter
    .Panels(3).AutoSize = sbrContents
  End With
End Sub
```

Tudi posamezni elementi vrstice stanja so novi predmeti, tokrat iz razreda Panel. Vsak predmet lahko vsebuje ročno vneseno besedilo ali pa prikazuje katero od določenih stanj, npr. trenutni čas, datum, stanje posebnih tipk in podobno.
V našem zgledu smo s samo vrstico stanja že dobili osnovno površino za izpis informacij. Več niti ne potrebujemo. No, zaradi ponazoritve dodajanja elementov v vrstico smo dodali še okvir za prikaz točnega časa in datuma. Vsak okvir pozna nekaj lastnosti, ki določajo njegov videz med izvajanjem. Predpišemo mu lahko minimalno širino, poravnavanje vsebine in njegovo prilagajanje širitvi oziroma krčenju gradnika, v katerem je vsebovan. Pri tem imamo na voljo nespremenljivo velikost, prilagajanje vsebini ali način »vzmeti« (sbrSpring), pri kateri se izbrani element samodejno razširi čez ves razpoložljivi prostor. Ta način smo uporabili za prvo polje, ura in datum pa bosta prikazana v okviru, katerega širina bo prilagojena dolžini, in bosta poravnana na sredino.

Za pravilen videz vmesnika moramo postoriti še nekaj: delovno površino obrazca si delijo štirje vsebniki, vsak je z lastnostjo Align potisnjen ob drug rob. V odzivu na dogodek Resize moramo zagotoviti, da bo slikovni okvir z delovno površino vedno raztegnjen čez ves razpoložljivi prostor:

```
' Dinamično določanje velikosti gradnikov
Private Sub Form_Resize()
  If ScaleWidth > 0 Then
    picRisba.Width = ScaleWidth - picOrodja.Width
  End If
End Sub
```

Tako, s tem smo prebrodili vso kodo vzpostavitev. Videz vmesnika si lahko ogledate na naslednji sliki:

Slika 10-28: Vzpostavljen grafični vmesnik Risarja

Datoteke sredstev

Z datotekami sredstev smo se že večkrat srečali. Tu si bomo natančneje ogledali njihove značilnosti, še posebej prednosti, ki nam olajšajo vzdrževanje in nadgrajevanje programov. Za ponazoritev bomo samo z orodji, ki spremljajo visual basic, pripravili potrebno datoteko sredstev za uporabniški vmesnik Risarja.

Vsebina

Datoteka sredstev (*resource file*) je posebna binarna datoteka, ki jo pripravi zunanji prevajalnik, *Resource Compiler*. V njej so najrazličnejši podatki, ki jih uporabimo pri izgradnji uporabniškega vmesnika. To so lahko:

- Bitne slike, ikone in kazalci miške.
- Znakovni nizi.

- Menuji in bližnjice.
- Opisi pogovornih oken.
- Večpredstavna vsebina (zvok, animacije).
- Poljubna druga binarna vsebina.

Ko program, ki uporablja zunanjo datoteko sredstev, prevedemo v izvršljivo obliko – datoteko .EXE, se skupaj s prevedeno kodo zapišejo tudi sredstva. Čeprav so shranjena v isti datoteki kakor izvršna koda, so neodvisna in jih lahko spreminjamo, ne da bi pokvarili programsko logiko. Če npr. vse programske izpise shranimo v datoteko sredstev, lahko program prilagodimo drugemu jeziku brez poseganja v izvorno kodo. Začasno izločimo sredstva, jih posodobimo in vnovič vgradimo v izvršljivo datoteko. Ker so sredstva povezana v izvršljivo datoteko, je olajšano tudi razpečevanje izdelanega programa. Odzivni časi programa, ki črpa lastna sredstva, so bistveno krajši kakor pri nalaganju podatkov iz ločenih zunanjih datotek.

NAMIG Za predelavo sredstev iz že prevedene datoteke potrebujemo poseben urejevalnik sredstev. Skupaj z visual basicom ne dobimo nič podobnega, na trgu programske opreme pa najdemo več tovrstnih orodij.

Orodja

Preden lahko zunanja sredstva uporabimo v visual basicu, moramo pripraviti posamezne vsebine in navodilo za prevajalnik (*resource script*) ter uspešno prevesti sredstva v datoteko s podaljškom .RES. Take datoteke lahko vključimo v poljuben projekt visual basica. V programski kodi so na voljo funkcije za črpanje posameznih sredstev.

Žal so orodja, priložena visual basicu, različic Professional in Enterprise, precej špartanska. Na voljo je precej drugih orodij, večinoma so na voljo ob razvojnih okoljih za jezik C++. Če smo srečni lastniki Microsoftovega skupka razvojnih okolij Visual Studio, lahko uporabimo izvrstna orodja tega okolja. No, za silo bo šlo tudi z orodji, ki jih imamo na voljo vsi uporabniki boljših različic visual basica. Najprej pa jih moramo najti.

- Urejevalnik bitnih slik najdemo na namestitveni plošči CD-ROM, v mapi \Tools\Imagedit\. Gre za program iz leta 1992, ki je že precej zastarel. Njegova glavna slabost je nezmožnost priprave ikon z več ločljivostmi, ki so v grafičnem okolju Oken 95 skorajda nujnost.
- Prevajalnik sredstev, Resource Compiler, se je najverjetneje namestil skupaj z visual basicom. Je v mapi Wizards v osnovni mapi visual basica. Njegova

sestavna dela sta izvršljiva datoteka Rc.exe in knjižnica Rcdll.dll. Prevajalnik sredstev je konzolni program, zato ga moramo uporabiti iz ukazne vrstice. Celotna oblika prevajalnika sredstev, skupaj z nekaj navodili, je tudi na plošči CD-ROM visual basica, v mapi \Tools\Resource\.

Izdelava datoteke sredstev

Sedaj potrebujemo izdelana sredstva in navodilo za prevajalnik. Oboje najdemo na plošči CD-ROM, ki spremlja to knjigo. Pogledati moramo v mapo \Primeri\2-Delavnica\3-Risar\2-Viri\Resource\. Vsebina te mape je naslednja:

```
COPY.BMP
HELP.BMP
NEW.BMP
OPEN.BMP
PRINT.BMP
REDO.BMP
SAVE.BMP
UNDO.BMP

CRTA.CUR
ELIPSA.CUR
OZNACI.CUR
PRAVOKOT.CUR
PROSTO.CUR

Risar.rc
```

Datoteke BMP so prevzete iz priloženih bitnih slik, ki spremljajo visual basic. So v mapi Graphics\Bitmaps\Offctlbr\Small\Color\, v osnovni mapi visual basica. Te bitne slike vsebujejo podobe gumbov za orodjarno, ki so enaki gumbom iz orodjarn programov v zbriki Microsoft Office. V mapi Graphics se skrivajo tudi druge ikone, ki smo jih uporabljali pri dosedanjih vajah.

Datoteke .CUR sem pripravil sam. Vsebujejo dvobarvne kazalce miške, ki jih bomo uporabili dvakrat: tako za grafično ponazoritev gumba kakor za kazalec miške.

Nazadnje potrebujemo še navodilo za prevajalnik, ki je navadna datoteka z neoblikovanim besedilom (*plain ASCII file*) in podaljškom .RC. Njena vsebina je naslednja:

```
1001 BITMAP NEW.BMP
1002 BITMAP OPEN.BMP
1003 BITMAP SAVE.BMP
1004 BITMAP PRINT.BMP
1005 BITMAP UNDO.BMP
1006 BITMAP REDO.BMP
1007 BITMAP COPY.BMP
1008 BITMAP HELP.BMP
1001 CURSOR OZNACI.CUR
1002 CURSOR PROSTO.CUR
1003 CURSOR CRTA.CUR
1004 CURSOR PRAVOKOT.CUR
1005 CURSOR ELIPSA.CUR
```

Vsakemu sredstvu moramo določiti enolično številsko oznako, njegovo vrsto in ime datoteke, v kateri je. Predpisana številka mora biti enolična za posamezno vrsto sredstva. Ne smemo uporabiti številke 1, ki jo interno uporablja sam visual basic. Pogosto je najbolje, če začnemo lastna sredstva označevati z dovolj visoko številko, kot je npr. 100 ali 1000. Če želimo izdelati datoteko sredstev, moramo pognati pozivnik MS-DOS, se prestaviti v mapo z zgoraj naštetimi datotekami in pognati prevajalnik sredstev:

```
>"d:\Program Files\DevStudio\VB\Wizards\RC.EXE" Risar.rc
>dir *.res
12.680  RISAR.RES
```

Pogosto je še laže, če si pred uporabo prepišemo datoteke Rc.exe in Rcdll.dll kar v isto mapo s sredstvi. Pri tem nam lahko celo pomaga Okenski komandant! Če v navodilih in pripravljenih datotekah ni bilo napak, nam prevajalnik sredstev izdela datoteko .RES, ki jo lahko vgradimo v projekt.

Če vam prevajanje sredstev ni uspelo, že izdelano datoteko najdete na priloženi plošči CD-ROM.

> **NAMIG** Podrobnejši opis izdelovanja datotek z navodili in prevajalnik sredstev so na voljo v elektronski pomoči, ki jo najdemo na plošči CD-ROM visual basica v mapi \Tools\Resource\.

Dodajanje datoteke sredstev projektu

Prevedeno datoteko sredstev moramo dodati v dejavni projekt. S tem bodo v njej shranjena sredstva postala dostopna v programski kodi. Dodajanje datoteke v projekt je enostavno:

V glavnem menuju **Project** izberemo ukaz **Add File**... in najdemo pripravljeno datoteko na disku. Enak ukaz je na voljo tudi v priročnem menuju projektnega okna. V posameznem projektu visual basica lahko uporabimo največ eno datoteko z zunanjimi sredstvi.

> **NAMIG** Ko datoteko s sredstvi vključimo v projekt, jo visual basic zaklene med izvajanjem ali prevajanjem programa. Zato moramo paziti, da je ne poizkušamo popravljati ravno v tem času.

Uporaba sredstev

Pripravljena in vključena sredstva uporabljamo glede na njihovo vsebino. V našem zgledu so v datoteki sredstev zgolj bitne slike, zato jih bomo črpali s funkcijo **LoadResPicture**. Pregled vseh funkcij za obravnavanje različnih sredstev je podan v naslednji tabeli:

Funkcija	Opis
LoadResString	Prebere niz, shranjen v datoteki sredstev.
LoadResPicture	Prebere bitno sliko, ikono ali kazalec iz datoteke sredstev.
LoadResData	Prebere poljubno sredstvo, ki ga vrne v obliki polja zlogov (*byte array*). Za branje slik je zato bolj priporočljiva uporaba funkcije LoadResPicture.

Ozrimo se še enkrat po dveh stavkih iz zgoraj zapisanih vzpostavitvenih podprogramov:

```
' Branje bitne slike (bitmap):
Set imgTmp = .Add(, , LoadResPicture(1005, vbResBitmap))

' Branje kazalca (cursor):
optLik(4).Picture = LoadResPicture(1005, vbResCursor)
```

Funkcija LoadResPicture zahteva enolično oznako posameznega sredstva in njegovo vrsto. Slika, ki jo vrne, je povsem enakovredna sliki, ki jo, denimo, vrne funkcija LoadPicture, le da je nalaganje sredstva lahko bistveno hitrejše, za povrh pa nam ni treba skrbeti, ali je datoteka sploh dostopna. Kakor vemo, so sredstva na koncu zapisana skupaj s kodo v izvršljivi datoteki oblike .EXE.

Branje niza je še preprostejše, saj funkcija LoadResString zahteva zgolj enolično oznako niza v datoteki sredstev.

NAMIG Kdor ne želi brskati po dokumentaciji za pripravo sredstev, lahko nize hitro vnese po naslednjih napotkih:

- Nize zapišemo v bloku `STRINGTABLE`
- Nize zapišemo neposredno v datoteko z navodili prevajalniku, med dvojnimi narekovaji.
- Dolžina posameznega niza ne sme presegati 255 znakov.

Oglejmo si datoteko z napotki (npr. `VIRI.RC`), ki vsebuje tudi nize:

```
1000 BITMAP SLIKA.BMP
STRINGTABLE
BEGIN
 1000 "Odpri datoteko"
 1001 "Shrani datoteko"
 1002 "Konec dela"
END
```

Zgled, razvit do te točke, najdete na priloženi plošči CD-ROM v mapi \Primeri\2-Delavnica\3-Risar\2-Viri\.

Programska koda

Nadaljujmo razvijanje Risarja. Uporabniški vmesnik sedaj zadovoljivo deluje, zato se moramo posvetiti osnovni funkcionalnosti programa. Najprej je treba določiti potrebno podatkovno strukturo, ki bo omogočala obnavljanje izrisane grafike v vsakem trenutku. Najuporabnejša podatkovna struktura za to je dinamično polje.

V to polje bi morali shranjevati najrazličnejše podatke, kot je:

- Izbrana barva.
- Izbrana debelina pisala.
- Izbrano risarsko orodje.
- Koordinate risanja.

Ker je barva določena z dolgim celim številom, se izkaže, da je za pomnjenje celotnega poteka izrisovanja najbolje uporabiti dinamično polje tipa Long. Z določenimi konstantami bomo vsako risarsko orodje predstavili v obliki celega števila, debelina pisala in koordinate v zaslonskih točkah pa so tako ali tako že cela števila.

Zaradi splošnosti in lažjega vzdrževanja kode bo vse izrisovanje kar največ potekalo skozi eno samo funkcijo. Zaradi sprotnega prikaza izbrane barve in debeline bomo uporabili dve modulski spremenljivki, ki bosta v vsakem trenutku vsebovali nazadnje izbrano barvo in debelino. Potrebovali bomo tudi modulsko spremenljivko, ki bo opisovala nazadnje izbrano risalno orodje.

Celotna podatkovna struktura glavnega obrazca je zapisana spodaj. Najprej moramo določiti dva lastna podatkovna tipa, zaradi preglednejšega zapisa koordinatnih parov in pravokotnega območja:

```
' Lastna tipa
' - koordinatni par
Private Type Tocka
  x As Integer
  y As Integer
End Type
' - okvir
Private Type Okvir
  P1 As Tocka
  P2 As Tocka
End Type
```

Vse kode risalnih orodij oziroma risalnih ukazov bomo združili v naštevnem tipu. Naštevni tip konstantnih vrednosti (*enumeration*) je novost visual basica 5. Omogoča nam, da skupino sorodnih celoštevilčnih konstantnih vrednosti določimo kot nov podatkovni tip. S tem izboljšamo berljivost kode, pri dosledni rabi tako določenega tipa pa nam lahko urejevalnik med vnašanjem programske kode že ponuja opisne vrednosti (podobno kakor npr. pri vnašanju parametrov za funkcijo MsgBox).

```
' Naštevne konstante
Private Enum RazpolozljiviUkazi
  ' - orodna ukaza
  ukazBarva = 1
  ukazDebelina = 2
  ukazProstoVmes = 3
  ukazProstoKonec = 4
  ' - ukazi za razveljavitev ( <0 )
  ukazProstoZacetek = -1
  ukazCrta = -3
  ukazPravokot = -4
  ukazElipsa = -5
  ukazZapolni = -6
End Enum
```

Kode ukazov, ki so manjše od nič, se bodo upoštevale pri razveljavitvi in uveljavitvi ukazov, pozitivne kode pa se bodo preskakovale. Natančen opis postopka razveljavitve in uveljavitve bomo spoznali malo pozneje.

Potrebujemo še nekaj običajnih konstant:

```
' Konstante
' - največje število shranjenih ukazov
Const constMaxUkazov = 10000
```

in konstante za opis stanja, v katerem je program. Večinoma so enakovredne izbranemu ukazu, posebnost je le stanje označevalca, ki ni risarski ukaz, temveč rabi za označevanje območja (*marquee*), ki ga lahko pozneje preslikamo na odložišče:

```
' - stanje izbranega orodja
Const stanjeOznacevalec = 0
Const stanjeProsto = ukazProstoZacetek
Const stanjeCrta = ukazCrta
Const stanjePravokot = ukazPravokot
Const stanjeElipsa = ukazElipsa
Const stanjeZapolni = ukazZapolni
```

Popoln opis stanja določajo naslednje spremenljivke:

```
' Stanje obrazca
' - opis orodja
Dim lngBarva As Long
Dim intDebelina As Integer
Dim intLik As Integer
' - stanja med delom
Dim blnRisanje As Boolean
Dim blnShranjeno As Boolean
' - ime datoteke
Dim strDatoteka As String
' - opis risbe
Dim intUkaz As Integer
Dim intRazveljavitve As Integer
Dim lngOpisRisbe() As Long
```

Potrebujemo še tri spremenljivke, ki bodo pomagale označevalcu pravokotnih območij:

```
' - označeno območje
Dim udtZacetek As Tocka
Dim udtKonec As Tocka
Dim udtClip As Okvir
```

Sedaj moramo ob vzpostavitvi obrazca poskrbeti še za začetne nastavitve vseh spremenljivk. Precej bomo morali dopolniti odziv na dogodek Load:

```
Private Sub Form_Load()

  ' Opis risbe, vzpostavitev (~32KB)
  ReDim lngOpisRisbe(constMaxUkazov)
  intUkaz = 0
  intRazveljavitve = 0
  lngBarva = vbBlack
  intDebelina = 1
  blnRisanje = False

  ' Vzpostavitev elementov vmesnika
  ' - gradniki Win95
  VzpostaviSlikanico
  VzpostaviGumbnico
  VzpostaviOrodjarno
  VzpostaviVrsticoStanja
  ' - pogovorno okno
  With dlgDialog
    .Filter = "Risarije (*.ris)|*.ris|Vse datoteke (*.*)|*.*"
    .DefaultExt = ".ris"
  End With

  ' Vzpostavitev označevanja
  ' - ponazoritev izbora celotne slike
  udtClip.P1.x = -1
  ' - ura za pomoč pri označevanju
  With tmrOznacevalec
    .Enabled = False
    .Interval = 250
  End With

  ' Vzpostavitev gradnika z risbo
  With picRisba
    .BackColor = vbWhite
    .DrawWidth = 1
    .MousePointer = 99
    .ScaleMode = vbPixels
  End With

  ' Zaključne vzpostavitve
  Caption = App.Title
  ' - koordinatni sistem
  ScaleMode = vbPixels
  ' - začetni položaj
  Move 0.05 * Screen.Width, 0.05 * Screen.Height, _
    0.8 * Screen.Width, 0.8 * Screen.Height

End Sub
```

Dinamično polje za shranjevanje ukazov bi sicer lahko povečevali v manjših korakih, vendar bi s tem program ne bil nič bolj učinkovit. V našem zgledu takoj zasežemo prostor za 10.000 ukazov, kar zahteva približno 32 KB pomnilnika. Če bi vzeli desetkrat manjšo količino, a bi jo med delom morali štirikrat ali petkrat povečati, bi imel visual basic precej dela s prepisovanjem podatkov in odzivnost programa bi bila lahko v tem trenutku okrnjena[39]. No, verjetno je danes že malo smešno razglabljati o nekaj tisoč zlogih pomnilnika, vendar hitrosti ni nikoli preveč.

Naslednja novost je vzpostavitev spremenljivke, ki opisuje označeno območje. Ker se koordinata –1 pri uporabi skale zaslonskih točk znotraj risalne površine med delom ne bo nikoli prikazala, bo ta vrednost ponazarjala, da ni označitve. Verjetno je najbolj smiselno, da izbrani ukaz »Kopiraj« brez posebej označenega dela slike prepiše kar vso sliko. Označevalcu bo pri delu pomagala še ura, ki jo vzpostavimo, vendar pustimo v stanju mirovanja (lastnost Enabled ni postavljena).

Za konec moramo poskrbeti še za risalno površino. Določimo ji belo barvo ozadja, koordinatni sistem v zaslonskih točkah, začetno debelino pisala ter nakažemo, da bomo nad risalno površino uporabljali lastne kazalce miške, določene v datoteki s sredstvi.

Nekaterih nastavitev ne moremo izvesti v odzivu na dogodek Load, zato obdelamo tudi dogodek Activate:

```
' Nekatere nastavitve so mogoče šele pri prikazanem obrazcu
Private Sub Form_Activate()
Static blnStorjeno
  ' Varovalka, samo prvič
  If Not blnStorjeno Then
    blnStorjeno = True
    intLik = stanjeProsto
    optLik(1).Value = True
    NovaRisba
  End If
End Sub
```

Spoznali smo že nevšečnosti, do katerih pride zaradi vnovičnih pojavitev dogodka Activate. Tu smo se zavarovali s statično spremenljivko, ki dopušča samo enkratno izvajanje naštetih stavkov. Z njimi poskrbimo za izbiro privzetega orodja. To je prostoročno risanje, ki ga opisuje ustrezna konstanta, pripada pa mu drugo orodje v stranski orodjarni (to ima zaporedno številko 1). Od tu naprej prepustimo vzpostavitev risbe novemu podprogramu, ki bo začel vsako novo risbo:

```
' Tu se začne vsaka nova risba
Sub NovaRisba()
  ' Brisanje morebitne vsebine
  picRisba.Cls
```

[39] Ukaz ReDim Preserve, ki poveča razpoložljivi prostor v polju, mora najprej zaseči nov kos pomnilnika in vanj prepisati obstoječe podatke.

```
  ' Nova je nepoimenovana
  frmGlavni.Caption = App.Title & " - [Nepoimenovana risba]"
  strDatoteka = ""
  blnShranjeno = True
  ' Začetne nastavitve risbe
  intUkaz = 0
  NastaviDebelino intDebelina
  NastaviBarvo lngBarva
End Sub
```

Ko začnemo novo risbo, najprej zbrišemo morebitno vsebino. Zatem označimo, da je risba nepoimenovana in da še nima sprememb, ki bi jih bilo treba shraniti. Vzpostavimo tudi modulsko dosegljivi kazalec zadnjega ukaza v spremenljivki intUkaz. Začetne nastavitve zapišemo v ukazno polje s posebej pripravljenima podprogramoma:

```
' Nastavi novo debelino in shrani ustrezen ukaz
Sub NastaviDebelino(intNovaDebelina As Integer)
  intDebelina = intNovaDebelina
  picRisba.DrawWidth = intDebelina
  lngOpisRisbe(intUkaz) = ukazDebelina
  lngOpisRisbe(intUkaz + 1) = intDebelina
  intUkaz = intUkaz + 2
End Sub

' Nastavi novo barvo in shrani ustrezen ukaz
Sub NastaviBarvo(lngNovaBarva As Long)
  lngBarva = lngNovaBarva
  picRisba.ForeColor = lngBarva
  lngOpisRisbe(intUkaz) = ukazBarva
  lngOpisRisbe(intUkaz + 1) = lngBarva
  intUkaz = intUkaz + 2
End Sub
```

Pri zgornjih dveh podprogramih smo prvič naleteli na shranjevanje ukazov risanja. Postopek je zelo preprost, paziti moramo le na dvoje:

- Kazalec zadnjega ukaza se mora povečevati natančno za toliko mest, kolikor smo jih porabili za zapis zadnjega ukaza.
- Vrednost modulskih spremenljivk mora vedno ustrezati izbranemu stanju med orodji.

Ponazoritev stanja

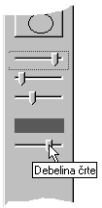
Oglejmo si sedaj, kako bomo uporabniku ponazorili izbrano orodje. V ta namen smo postavili posebno navpično orodjarno, v kateri imajo gumbi na svoji površini kar bodoči kazalec miške. Zakaj pa ne? Oboje je bitna slika. Za izbiranje barve in debeline je nižje še skupina drsnikov. Med njimi je ravno primeren prostor za prikaz trenutno izbrane debeline in barve.

Aha, zdaj vemo, zakaj smo orodjarno postavili znotraj slikovnega okvira. Ta ni navaden vsebnik (kot npr. okvir), ampak omogoča tudi uporabo grafičnih metod. Izbrano barvo in debelino bomo torej preprosto grafično izrisali v prostor med skupinama drsnikov.

Upam, da se še vsi spominjamo težav, ki nas lahko doletijo pri uporabi grafičnih metod. Pri programiranju moramo stremeti k temu, da vse grafične metode uporabljamo zgolj v odzivu na dogodek Paint. Pri stranski orodjarni bomo storili natančno to.

```
' Nakaži barvo in debelino v orodjarni
Private Sub picOrodja_Paint()
  picOrodja.Line (8, sldNastavitve(3).Top - 6)-Step(50, -(intDebelina - 1)), _
    lngBarva, BF
End Sub
```

Slika 10-29: Ponazoritev izbrane barve in debeline je izrisana z grafično metodo Line.



Vse skupaj postori klic metode Line. Izhodišče risanja izračunamo na osnovi položaja zadnjega drsnika. Trenutno izbrana barva in debelina je vedno na voljo v modulsko dosegljivih spremenljivkah, ki ju uporabimo za izris pravokotnika.

Izbira orodja

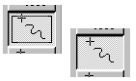
Izbiranje orodja rešimo v odzivnih procedurah ustreznih gradnikov. Najprej si oglejmo kodo, ki poskrbi za izbor risalnega orodja:

```
' Izbira orodja
Private Sub optLik_Click(Index As Integer)
  ' Spremeni kazalec miške nad risbo
  picRisba.MouseIcon = optLik(Index).Picture
  ' Zapomni si izbrano orodje
  Select Case Index
    Case 0
      intLik = stanjeOznacevalec
    Case 1
      intLik = stanjeProsto
    Case 2
      intLik = stanjeCrta
    Case 3
      intLik = stanjePravokot
    Case 4
      intLik = stanjeElipsa
```

```
    Case Else
      Debug.Print "NEZNAN LIK!"
  End Select
  ' Odpravi označitev vnosne točke
  picRisba.SetFocus
End Sub
```

Vsa stikala so v polju gradnikov, zato jih obdelamo hkrati. Ker je slika na stikalu pravzaprav kazalec (izvira iz datoteke .CUR), jo lahko neposredno uporabimo za kazalec miške nad risalno površino. Zatem s stavkom Select Case shranimo izbrano orodje v še eno modulsko spremenljivko, intLik, ki v vsakem trenutku označuje trenutno izbrano orodje. Zaradi večje trdoživosti programa obravnavamo tudi nepravilne vrednosti. Uporaba predmeta Debug nam zagotavlja, da bodo morebitna sporočila vidna le med preizkušanjem programa v razvojnem okolju, v končnem izdelku pa jih ne bo.

Slika 10-30: Nadležnega označevanja vnosne točke se lahko znebimo z zvijačo.



Na koncu uporabimo še manjšo zvijačo. Ko smo kliknili stikalo, se je vnosna točka samodejno preselila nanj. Ponazorjena je z nekoliko nadležnim okvirom, ki se izriše vrh grafike na gumbu. Zaradi tega vnosno točko nemudoma oddamo nazaj risalni površini.

Izbira barve in debeline

Tudi pri izboljšanih drsnikih imamo na voljo dva dogodka: Scroll se pojavlja *med* spreminjanjem vrednosti drsnika, Change pa na koncu. Vemo že, da je najbolje uporabiti oba:

```
' Reakcija na izbiranje barve/debeline
Private Sub sldNastavitve_Scroll(Index As Integer)
  If Index < 3 Then
    ' - zgornji drsniki določajo barve
    lngBarva = RGB(sldNastavitve(0).Value, _
      sldNastavitve(1).Value, sldNastavitve(2).Value)
  Else
    ' - spodnji pa debelino
    intDebelina = sldNastavitve(Index).Value
  End If
  ' Osveži prikaz izbranega orodja
  picOrodja.Cls
  picOrodja_Paint
End Sub
```

```
' Ob koncu spreminjanja storimo enako kakor med njim
Private Sub sldNastavitve_Change(Index As Integer)
  sldNastavitve_Scroll (Index)
End Sub
```

S stavkom If preverimo, kaj uporabnik pravzaprav spreminja, saj so vsi štirje drsniki v istem polju gradnikov. Spremembo vestno zapišemo, vendar je ne vpisujemo na seznam ukazov, saj se bo barva pred risanjem lahko še večkrat spremenila. Dolžni smo le obnoviti prikaz izbrane barve in debeline. To opravilo je enostavno: zbrišemo lastno grafiko in ročno zahtevamo osvežitev s klicem procedure za dogodek Paint.

NAMIG Neposreden klic odzivne procedure za dogodek Paint ni ravno najbolj v skladu s pravili, ki smo jih obravnavali pri uporabi grafičnih metod. Tu nas rešuje privzeta nastavitev lastnosti ClipControls, ki je resnična, zato se obnavljanje slike vedno izogiba gradnikom na zgornjem sloju. Za preizkus poskusite spremeniti vrednost te lastnosti za gradnik picOrodja in opazili boste nevšečno utripanje vsebine ob vsaki spremembi barve ali debeline.

Pravilnejši pristop bi bila uvedba dodatnega gradnika slikovni okvir (Picture Box), ki bi bil namenjen samo ponazoritvi stanja barve in debeline. S tem bi izrisovanje fizično omejili v ločen gradnik. Vse risanje bi še vedno ostalo znotraj metode Paint, vendar bi lahko učinkovito osveževanje gradnika dosegli s klicem metode Refresh. Metoda Refresh obnovi vsebino gradnika, tako da najprej sproži dogodek Paint, zatem pa izriše še morebitne gradnike na višjih slojih.

In zakaj metode Refresh ne uporabimo tudi v zgornjem zgledu? Ker rišemo po površini, v kateri so še dodatni zunanji gradniki (novi drsniki). Tudi tu lahko napravimo preizkus. Namesto neposrednega klica procedure Paint pokličemo metodo Refresh gradnika picOrodja. Ob vsaki spremembi bomo opazili nadležno utripanje drsnikov. Zapisana koda je torej kompromis, ki zagotavlja manjšo porabo sistemskih sredstev (prihranili smo en gradnik slikovni okvir) in dovolj varno izrisovanje uporabniške grafike.

Risanje likov

Sedaj si oglejmo, kako spremljamo uporabnika med risanjem po površini gradnika picRisba. Risanje se izvaja izključno z miško, zato moramo obravnavati dogodke MouseDown, MouseMove in MouseUp. Ti trije dogodki obravnavajo miško na *nizki* stopnji, saj njihove odzivne procedure za razliko od dogodka Click prejmejo dodatne parametre, ki opisujejo trenutni položaj kazalca miške in stanje posebnih tipk. Najprej si oglejmo začetek risanja, ki ga sproži dogodek MouseDown:

```
Private Sub picRisba_MouseDown(Button As Integer, Shift As Integer, _
x As Single, y As Single)

  ' Leva tipka prične risanje
  If Button = vbLeftButton Then

    ' Zapišemo stanje, ki ga bomo uporabili
    NastaviBarvo lngBarva
    NastaviDebelino intDebelina

    ' Označevalec utripa
    If tmrOznacevalec.Enabled Then
      ObnoviRisbo
    End If

    blnRisanje = True

    udtZacetek.x = x
    udtZacetek.y = y
    udtKonec.x = x
    udtKonec.y = y

    If intLik <> stanjeOznacevalec Then
      blnShranjeno = False
      lngOpisRisbe(intUkaz) = intLik
      lngOpisRisbe(intUkaz + 1) = x
      lngOpisRisbe(intUkaz + 2) = y
      intUkaz = intUkaz + 3
    End If

    If intLik <> stanjeProsto Then
      ' Nastavi prerisovanje v načinu XOR
      picRisba.DrawMode = vbXorPen
      picRisba.DrawStyle = vbDot
      picRisba.DrawWidth = 1
    End If

  ' Druga tipka prikaže priročni menu
  Else

    PopupMenu mnuPrirocni, vbPopupMenuRightButton, , , mnuPrirocniUkaz(0)

  End If
End Sub
```

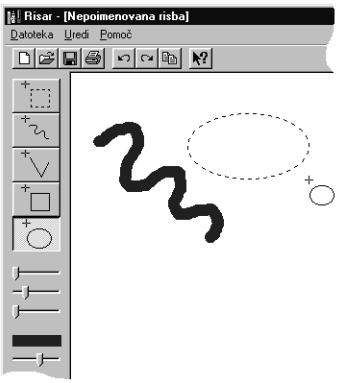
Ko je uporabnik nad risalno površino stisnil gumb miške, se pokliče zgornja procedura. Če gre za levi gumb, začnemo uporabljati izbrano orodje. Najprej dokončno zapišemo trenutno izbrano barvo in debelino. Vse vmes opravljene spremembe so pozabljene, saj se niso uporabile za risanje likov. Pomembno je, da zapišemo barvo in debelino, s katero se bo lahko pravkar začelo izrisovati.

Pred izrisovanjem moramo izklopiti morebitni dejavni označevalec (*marquee*), ki ga ponazarja dejavni gradnik ura. Če je označevalec slučajno dejaven, enostavno obnovimo risbo. Ustrezna procedura ObnoviRisbo v vsakem primeru izklopi uro in vnovič nastavi spremenljivke, ki vplivajo na izrisovanje. S spremenljivko blnRisanje označimo, da smo prešli na risanje, ki je zelo pomembno pri obravnavi dogodka

MouseMove. Zapomniti si moramo tudi trenutni koordinatni par. Če je izbrano risalno orodje in ne označevalec, označimo, da bo na risbi prišlo do spremembe, in zapišemo trenutno izbrano orodje in par koordinat v ukazno polje.

Naslednje preverjanje ugotavlja, ali gre za prostoročno risanje. V tem primeru bo miška puščala trajno sled, sicer pa moramo uporabniku zgolj *nakazovati*, kakšen lik se bo izrisal. Pri nakazovanju uporabimo poseben slog izrisovanja:

- Slog pisala je črtkan, zato mora biti debelina 1.
- Bitna operacija s podlago je XOR (različno), kar omogoča poznejše obnavljanje vsebine brez posledic.



Slika 10-31: Prostoročno risanje pušča sled, pri risanju likov pa samo nakazujemo bodoči videz lika.

V odzivu na dogodek MouseDown lahko nastavimo modulsko spremenljivko blnRisanje. Če ima resnično vrednost, je to za odzivno proceduro dogodka MouseMove znak, da mora ob vsakem premiku miške izrisovati vsebino, ustrezno izbranemu orodju:

```
Private Sub picRisba_MouseMove(Button As Integer, Shift As Integer, _
x As Single, y As Single)

  If blnRisanje And (x <> udtKonec.x Or y <> udtKonec.y) Then

    If intLik = stanjeProsto Then
      ' Stanje koordinat
      StatusBar1.Panels(1).Text = "[ " & x & " , " & y & " ]"

      'Izrišem odsek črte
      RisiLik intLik, udtKonec.x, udtKonec.y, (x), (y), lngBarva

      udtKonec.x = x
      udtKonec.y = y

      'Shranim ukaz
      lngOpisRisbe(intUkaz) = ukazProstoVmes
      lngOpisRisbe(intUkaz + 1) = x
      lngOpisRisbe(intUkaz + 2) = y
      intUkaz = intUkaz + 3
```

```
    Else
      ' Stanje koordinat
      StatusBar1.Panels(1).Text = "[ " & udtZacetek.x & _
        " , " & udtZacetek.y & " - " & x & " , " & y & " ]"

      RisiLik intLik, udtZacetek.x, udtZacetek.y, _
        udtKonec.x, udtKonec.y, vbWhite
      udtKonec.x = x
      udtKonec.y = y
      RisiLik intLik, udtZacetek.x, udtZacetek.y, _
        udtKonec.x, udtKonec.y, vbWhite

    End If

  Else
    ' Stanje koordinat
    StatusBar1.Panels(1).Text = "[ " & x & " , " & y & " ]"
  End If

End Sub
```

Pogoj za risanje je nastavljena spremenljivka blnRisanje, poleg tega pa pričakujemo, da uporabnik z miško križari po točkah, različnih od izhodišča risanja. To preverjanje je potrebno zaradi poznejšega preračunavanja pri risanju elips. Če bi dopustili vrnitev na izhodiščno koordinato, bi prišlo do napake prekoračitve (*overflow*) pri izračunu faktorja sploščenosti.

Posebnost pri risanju je prostoročno risanje. Tu moramo v ukazni niz zapisati prav vsak zaznani premik miške. Vsi drugi elementi se bodo dokončno izrisali šele, ko bo uporabnik miško izpustil. Zato druge like rišemo dvakrat. Prvič na starih koordinatah. Zaradi bitne operacije XOR bo vnovično risanje popolnoma obnovilo ozadje. Po tem »brisanju« označitve prestavimo ciljni koordinati in izrišemo novo označitev. Odziv na dogodek MouseMouse uporabimo tudi za osvežitev vrednosti koordinat v vrstici stanja.

Vsako risanje se konča z izpustom miške:

```
Private Sub picRisba_MouseUp(Button As Integer, Shift As Integer, x As Single, _
y As Single)
  If Button = vbLeftButton And blnRisanje Then

    blnRisanje = False

    If intLik = stanjeProsto Then
      ' Shranim ukaz
      lngOpisRisbe(intUkaz) = ukazProstoKonec
      intUkaz = intUkaz + 1
      intRazveljavitve = 0

    Else
      ' Odstranim oznako
      RisiLik intLik, udtZacetek.x, udtZacetek.y, _
        udtKonec.x, udtKonec.y, vbWhite

      ' Izrišem lik
      udtKonec.x = x
      udtKonec.y = y
```

```
      If intLik <> stanjeOznacevalec Then
        picRisba.DrawStyle = vbSolid
        picRisba.DrawMode = vbCopyPen
        picRisba.DrawWidth = intDebelina

        RisiLik intLik, udtZacetek.x, udtZacetek.y, _
          udtKonec.x, udtKonec.y, lngBarva

        ' Shranim ukaz
        lngOpisRisbe(intUkaz) = udtKonec.x
        lngOpisRisbe(intUkaz + 1) = udtKonec.y
        intUkaz = intUkaz + 2
        intRazveljavitve = 0

      Else
        RisiLik intLik, udtZacetek.x, udtZacetek.y,
          udtKonec.x, udtKonec.y, vbWhite
        udtClip.P1 = udtZacetek
        udtClip.P2 = udtKonec
        picRisba.DrawStyle = vbSolid
        tmrOznacevalec.Enabled = True

      End If
    End If

  End If

End Sub
```

Če smo doslej risali, so pri izpustu miške tri možnosti:

- Pri prostoročnem risanju v ukazno polje dodamo oznako za konec risanja.
- Pri drugih likih odstranimo še zadnjo oznako ter dokončno izrišemo lik v pravi barvi in s pravo debelino. Risanje označimo tudi v ukaznem polju.
- Pri označevalcu izrišemo lik znova, a tokrat s polno črto. Bitno operacijo XOR ohranimo zaradi učinka animacije. Zapišemo koordinatni par označenega območja in vključimo uro. Ura bo poskrbela za animirani videz označevanja (*rubber band*).

Vsa izrisovanja smo posredovali lastnemu podprogramu RisiLik. Oglejmo si njegovo kodo:

```
' Osnovni podprogram risanja
Private Sub RisiLik(intLik As Integer, X1 As Integer, Y1 As Integer, _
X2 As Integer, Y2 As Integer, lngBarva As Long)
  Select Case intLik
    Case stanjeProsto
      ' - prostoročno
      picRisba.Line (X1, Y1)-(X2, Y2), lngBarva
    Case stanjeCrta
      ' - ravne črte
      picRisba.Line (X1, Y1)-(X2, Y2), lngBarva
    Case stanjePravokot, stanjeOznacevalec
      ' - pravokotnik in označevalec
      picRisba.Line (X1, Y1)-(X2, Y2), lngBarva, B
```

```
    Case stanjeElipsa
      ' - elipsa
      If X1 <> X2 Then picRisba.Circle ((X1 + X2) \ 2, (Y1 + Y2) \ 2), _
        Abs(X2 - X1) \ 2, lngBarva, , , (Y2 - Y1) / (X2 - X1)
    Case Else
      ' - sicer neznan ukaz
      Debug.Print "NEVELJAVEN LIK: " & intLik
  End Select
End Sub
```

Tudi ta podprogram ima vgrajeno opozorilo, ki nas samo v razvojnem okolju opozarja na morebitne neveljavne like. Podprogram je zanimiv predvsem zaradi obširnega preračunavanja, ki ga potrebujemo, da iz podanih koordinat izračunamo ustrezne parametre za izris elipse.

Kako dosežemo animirani učinek obrobe označevalca? Zelo preprosto:

```
Private Sub tmrOznacevalec_Timer()
  RisiLik stanjeOznacevalec, udtZacetek.x, _
    udtZacetek.y, udtKonec.x, udtKonec.y, vbWhite
End Sub
```

Z uro v rednih časovnih intervalih v bitnem načinu XOR prerisujemo črtkano črto s polno. Zato se zdi, kakor bi obroba utripala, oziroma potovala naokoli.

Obnavljanje

Med risanjem smo v ukazno polje vestno zapisali vsak izrisani element. Obnova celotne risbe bi ne smela biti pretežka. Zanjo skrbi lasten podprogram ObnoviRisbo:

```
' Obnovitev risbe po shranjenih ukazih
Sub ObnoviRisbo()
Dim lngShraniBarvo As Long
Dim intShraniDebelino As Integer
Dim i As Integer
Dim x As Integer
Dim y As Integer

  ' Označi morebitno čakanje
  Screen.MousePointer = vbHourglass

  ' Shrani nastavitve
  lngShraniBarvo = lngBarva
  intShraniDebelino = intDebelina

  ' Izključi morebitno označitev
  If tmrOznacevalec.Enabled Then
    tmrOznacevalec.Enabled = False
    picRisba.DrawMode = vbCopyPen
  End If

  ' Zbriši okno
  picRisba.Cls
```

```
  ' Obnovi risbo
  i = 0
  Do
    Select Case lngOpisRisbe(i)
      Case ukazBarva
        lngBarva = lngOpisRisbe(i + 1)
        i = i + 2

      Case ukazDebelina
        intDebelina = lngOpisRisbe(i + 1)
        picRisba.DrawWidth = intDebelina
        i = i + 2

      Case ukazProstoZacetek
        udtZacetek.x = lngOpisRisbe(i + 1)
        udtZacetek.y = lngOpisRisbe(i + 2)
        i = i + 3

      Case ukazProstoVmes
        x = lngOpisRisbe(i + 1)
        y = lngOpisRisbe(i + 2)
        picRisba.Line (udtZacetek.x, udtZacetek.y)-(x, y), lngBarva
        udtZacetek.x = x
        udtZacetek.y = y
        i = i + 3

      Case ukazProstoKonec
        i = i + 1

      Case ukazCrta, ukazPravokot, ukazElipsa
        udtZacetek.x = lngOpisRisbe(i + 1)
        udtZacetek.y = lngOpisRisbe(i + 2)
        udtKonec.x = lngOpisRisbe(i + 3)
        udtKonec.y = lngOpisRisbe(i + 4)
        RisiLik (lngOpisRisbe(i)), udtZacetek.x, udtZacetek.y, udtKonec.x, _
          udtKonec.y, lngBarva
        i = i + 5

      Case Else
        Debug.Print "NEZNAN UKAZ RISANJA: " & lngOpisRisbe(i)
        Exit Do

    End Select

  Loop Until i = intUkaz

  ' Uveljavi izbrane nastavitve
  lngBarva = lngShraniBarvo
  intDebelina = intShraniDebelino

  ' Delo je končano
  Screen.MousePointer = vbDefault

End Sub
```

Vsako obnavljanje risbe poteka od vrha navzdol, prek vseh shranjenih ukazov risanja. Najprej moramo shraniti obstoječe stanje, ki ga bomo po koncu risanja obnovili. Zatem preverimo morebiti vključeni označevalec, ki pomeni še naprej

dejavno bitno operacijo XOR. Pred začetkom obnavljanja risbe zbrišemo vso uporabniško grafiko. S tem tudi dokončno odstranimo morebitni označevalec in druge nezaželene ostanke. Najvažnejši del podprograma je zanka, v kateri se sprehajamo po shranjenih ukazih. Samo risanje preložimo na ramena funkcije RisiLik, v zanki pa moramo skrbeti, da natančno povečujemo kazalec naslednjega ukaza. Ko ta dohiti trenutno nastavljeni kazalec zadnjega ukaza, zanko končamo. Na koncu še obnovimo stanje in vnovič nastavimo kazalec miške.

Ta podprogram moramo poklicati ob vsaki potrebi obnovitve vsebine gradnika picRisba:

```
' Obnovitev risbe
Private Sub picRisba_Paint()
  ObnoviRisbo
End Sub
```

S tem je uporabniško izrisana grafika v gradniku picRisba povsem obstojna, brez dodatnih potreb po pomnilniku (lastnost AutoRedraw je ves čas izključena).

Križišča

Pri razvejenih uporabniških vmesnikih, kot je tudi Risarjev, se izplača čim več zaokroženih opravil zapisati v lastnih podprogramih in ne neposredno znotraj odzivnih procedur, saj isto opravilo lahko izberemo z glavnega menuja, orodjarne, priročnega menuja in morda še od kod. Zato moramo v programski kodi zapisati še nekaj vejitvene kode, ki izbere pravi podprogram na podlagi izbranega ukaza ali gumba.

Glavni menu

Na ukaze glavnega menuja se odzivamo s preprostimi procedurami, ki delo preložijo na ramena posebej razvitih podprogramov. Odzivi na ukaze menuja Datoteka so:

```
' Menujski ukazi
Private Sub mnuDatotekaNova_Click()
  If OpozoriloSprememb Then Exit Sub
  NovaRisba
End Sub

Private Sub mnuDatotekaOdpri_Click()
  If OpozoriloSprememb Then Exit Sub
  OdpriDatoteko
End Sub

Private Sub mnuDatotekaShrani_Click()
  ShraniDatoteko strDatoteka
End Sub
```

```
Private Sub mnuDatotekaShraniKot_Click()
  ShraniDatoteko ""
End Sub

Private Sub mnuDatotekaIzvoziKotBitnoSliko_Click()
  IzvoziKotBitnoSliko
End Sub
```

Za razliko od Urejevalnika smo pri Risarju tudi za opozorilo ob morebitni zavržbi vseh sprememb zapisali ločeno funkcijo OpozoriloSprememb, ki vrne resnično vrednost le, če se uporabnik strinja z izgubo podatkov. Koda funkcije ni nič posebnega:

```
' Opozorilo pred zavržbo spremenjene risbe
Private Function OpozoriloSprememb()
Dim intIzid As Integer
  If Not blnShranjeno Then
    intIzid = MsgBox("Risba je bila spremenjena, shranim spremembe?", _
      vbYesNoCancel + vbExclamation)
    If intIzid = vbYes Then
      mnuDatotekaShrani_Click
      OpozoriloSprememb = False
    ElseIf intIzid = vbCancel Then
      OpozoriloSprememb = True
    End If
  End If
End Function
```

Funkcija najprej preveri, ali je na risbi sploh prišlo do kake spremembe. V tem primeru prikažemo opozorilo. Če se uporabnik premisli, vrnemo neresnično vrednost. Če želi spremembe shraniti, posnemamo izbiro menujskega ukaza Shrani.

Vrstica gumbov

Naslednja vejitev izvajanja je odziv na klik v vrstici gumbov. Gumbi v njej so pravzaprav le bližnjice do enakovrednih menujskih ukazov. To je razvidno tudi iz odzivne procedure:

```
' Križišče ob kliku gumba v vrstici
Private Sub Toolbar1_ButtonClick(ByVal Button As Button)
  Select Case Button.Key
    Case "Nova"
      mnuDatotekaNova_Click
    Case "Odpri"
      mnuDatotekaOdpri_Click
    Case "Shrani"
      mnuDatotekaShrani_Click
    Case "Razveljavi"
      mnuUrediRazveljavi_Click
    Case "Uveljavi"
      mnuUrediUveljavi_Click
    Case "Kopiraj"
      mnuUrediKopiraj_Click
```

```
    Case "Pomoc"
      mnuPomocPojasnila_Click
  End Select
End Sub
```

Odzivna procedura prejme parameter, ki predstavlja pritisnjeni gumb kot predmet razreda Button. V vejitvenem stavku Select Case se sprašujemo po ključu pritisnjenega gumba. Ta ključ določimo ob vzpostavitvi gradnika, ko dodajamo nove predmete Button z metodo Add. Smiselna opisna imena gumbov znatno izboljšajo berljivost kode.

Priročni menu

Tudi priročni menuji (*context menus*) so pomemben del uporabniškega vmesnika, ki se zgleduje po Oknih 95. Če bi želeli zadostiti strogim pravilom za pridobitev zaščitnega znaka »Windows 95 compatible«, bi morali priročni menu prirediti prav vsakemu izstopajočemu elementu vmesnika. Mi smo ga pripravili za risalno površino.

Nad risalno površino se zgodi dogodek MouseDown, tudi če uporabnik pritisne desno tipko miške. Tu ne bomo ničesar risali, temveč prikažemo priročni menu, ki je določen z nevidnim menujem mnuPrirocni. Oglejmo si še enkrat izsek programske kode v odzivni proceduri:

```
Private Sub picRisba_MouseDown(Button As Integer, Shift As Integer, _
x As Single, y As Single)

  ' Leva tipka sproži risanje
  If Button = vbLeftButton Then

  ( ... )

  ' Druga tipka prikaže priročni menu
  Else

    PopupMenu mnuPrirocni, vbPopupMenuRightButton, , , mnuPrirocniUkaz(0)

  End If
End Sub
```

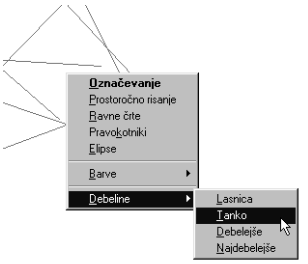
Z metodo PopUpMenu smo se že srečali, tu pa jo uporabljamo še z dvema posebnima nastavitvama. Konstanta vbPopupMenuRightButton pomeni, da lahko uporabnik še naprej izbira po menuju z desno tipko, leva tako ali tako vedno deluje. Zadnji parameter opisuje menujski ukaz, ki je izpisan krepko. Pri nas je to ukaz, ki izbere označevanje. Ker se bo označevanje izbralo tudi pri dvakratnem kliku po risalni površini, je ta izbira na neki način privzeta[40]. Dvakratni klik obravnavamo v posebni odzivni proceduri:

[40] Podobno je v Raziskovalcu. Ukaz, ki je v priročnem meniju izpisan krepko, se uporabi tudi, če po datoteki dvakrat kliknem.

```
' Dvakratni klik naj pomeni isto kot krepko izpisana
' možnost v priročnem menuju
Private Sub picRisba_DblClick()
  mnuPrirocniUkaz_Click 0
End Sub
```

Enostavno simuliramo izbiro privzetega ukaza.

Slika 10-32: Priročni menu risalne površine



Oglejmo si odzivne procedure za menujske ukaze priročnega menuja. Obravnavati moramo tri skupine ukazov. V prvi so ukazi, ki izberejo ustrezno orodje:

```
' Križišče za priročni menu
Private Sub mnuPrirocniUkaz_Click(Index As Integer)
  optLik(Index).Value = True
  optLik_Click Index
End Sub
```

V odzivni proceduri moramo zagotoviti povsem enak učinek, kakor da bi uporabnik pritisnil na enakovreden gumb v orodjarni. To bi sicer lahko dosegli zgolj s klicem njegove odzivne procedure, vendar bi videz vmesnika ostal neusklajen. Zato pred klicem procedure še ročno »ugreznemo« pripadajoči gumb, ki je v bistvu opcijsko stikalo.

Sledi določanje barve:

```
' Izbira barve v priročnem menuju
Private Sub mnuPrirocniBarveIzbira_Click(Index As Integer)
Dim intR As Integer
Dim intG As Integer
Dim intB As Integer

  Select Case Index
    Case 0 ' Rdeča
      intR = 255
      intG = 0
      intB = 0

    Case 1 ' Rumena
      intR = 255
      intG = 255
      intB = 0
```

```
    Case 2 ' Zelena
      intR = 0
      intG = 255
      intB = 0

    Case 3 ' Modra
      intR = 0
      intG = 0
      intB = 255

  End Select

  sldNastavitve(0).Value = intR
  sldNastavitve(1).Value = intG
  sldNastavitve(2).Value = intB

End Sub
```

Tudi pri določanju barve moramo zagotoviti usklajenost drsnikov z izbrano barvo. Zato vsako barvo določimo v odtenkih osnovnih barv, nato pa z njimi neposredno spremenimo vrednosti drsnikov. To je tudi vse, saj sprememba drsnikov povzroči dogodek Change, ki poskrbi za osvežitev prikazanega stanja v orodjarni.

Za konec še sprememba debeline:

```
' Izbira debeline v priročnem menuju
Private Sub mnuPrirocniDebelineIzbor_Click(Index As Integer)
With sldNastavitve(3)
    Select Case Index
      Case 0 ' lasnica
        .Value = .Min

      Case 1 ' tanko
        .Value = Int((.Max - .Min) * 0.15)

      Case 2 ' debelejše
        .Value = Int((.Max - .Min) * 0.5)

      Case 3 ' najdebelejše
        .Value = .Max

    End Select
  End With
End Sub
```

Nič posebnega, zanimivejše je le določanje debeline. Pri tem se ne opiramo na konstantne vrednosti, ampak debelino izračunamo. Tako je program laže spreminjati, saj ob drugačnem razponu debelin zgolj spremenimo vzpostavitev zadnjega drsnika.

Delo z datotekami

Risbe bomo shranjevali v posebno obliko *binarne* datoteke. Za tak korak smo se odločili zaradi značilnosti podatkovne strukture, ki opisuje risbo. Shranjevanje v bitno sliko bi ne bilo smiselno, saj bi bile izhodne datoteke prevelike in za nadaljnje delo popolnoma neuporabne. Zaradi uporabe binarnih datotek bomo vsako

shranjeno sliko lahko v celoti obvladovali, vključno z razveljavljanjem shranjenih ukazov vse do samega začetka risanja.

Datotečna čarovnija

Pri uporabi lastnega zapisa datotek je datoteko koristno opremiti z glavo, ki opisuje njeno poreklo in vsebino. Operacijski sistem Okna na žalost le izvršljive datoteke analizira na podlagi njihove vsebine. Povsod drugod se opira na končnico. To uporabnikom lahko povzroči hude preglavice, saj nam nihče ne zagotavlja, da je npr. v datoteki s končnico .vbp res projekt visual basica. To ugotovimo šele po uspešno ali neuspešno odprti datoteki[41]. No, pustimo te težave drugim. Sami bomo poskrbeli, da bodo naše podatkovne datoteke ustrezno podpisane.

Podpis datoteke bo vračala posebna funkcija DatotecnaCarovnija:

```
' Funkcija, ki določa enolično oznako datoteke
Private Function DatotecnaCarovnija() As String
  DatotecnaCarovnija = "RS" & Chr(0) & _
    "Risarjeva risba - RISAR (c) 1997 UM" & Chr(26)
End Function
```

Za podpis bi lahko uporabili tudi konstanten niz, vendar bi vanj težko vnesli posebna znaka, ki ju vrača funkcija **Chr**. Funkciji posredujemo kodo znaka, ki ga želimo. Mi smo priklicali neveljaven znak NULL (koda 0) in znak za konec datoteke, EOF (koda 26).

> **NAMIG** Ime podprograma izvira iz računalniškega žargona. Izraz *File Magic* predstavlja enolično zaporedje znakov na začetku datoteke, ki opredelijo njeno vrsto. Izvršljive datoteke .EXE, .DLL, .DRV in še nekatere imajo znaka MZ, stisnjene datoteke .ZIP znaka PK, slike BMP znaka BM itn. Risar bo uporabil zaporedje RS.
>
> Podpis binarne datoteke pogosto obsega še berljiv opis vsebine (*human readable form*), ki vključuje niz znakov, končan z znakom za konec datoteke. Če bi uporabnik po pomoti želel tovrstno datoteko odpreti kot besedilo ali jo izpisati s tiskalnikom ali na konzolo, bi dobil samo berljivo pojasnilo, zatem pa bi se branje datoteke končalo. Ker bomo datoteko odprli v binarnem načinu, je znak EOF za nas zgolj še ena binarna koda.

Shranjevanje

Pri razvoju Urejevalnika smo zapisali različen podprogram za prikaz standardnega pogovornega okna in shranjevanje datoteke. Vzroki za to so bili natančneje opisani.

[41] Nekateri operacijski sistemi vrsto datoteke ugotavljajo izključno na podlagi njene vsebine in atributov.

V našem zgledu je vsa programska koda zbrana v enem obrazcu, zato bomo shranjevanje datoteke pod znanim in neznanim imenom, skupaj z morebitnim prikazom pogovornega okna, opravili na enem mestu:

```
' Shranjevanje datoteke
Private Function ShraniDatoteko(strImeDatoteke As String)
Dim intDatoteka As Integer

  If strImeDatoteke = "" Then
    With dlgDialog
      .DialogTitle = "Shrani risbo"
      .Flags = cdlOFNPathMustExist + cdlOFNOverwritePrompt + cdlOFNHideReadOnly
      .filename = strDatoteka
      .CancelError = True

' Prvi prestreznik skrbi za morebitni preklic
On Error GoTo NapakaPreklic
      .ShowSave
      strImeDatoteke = .filename
    End With
  End If

' Od tod naprej skrbimo za datotečne napake
On Error GoTo NapakaDatoteka
  ' Morebiti obstoječo datoteko moramo zbrisati
  If Dir(strImeDatoteke) <> "" Then Kill strImeDatoteke

  intDatoteka = FreeFile
  Open strImeDatoteke For Binary As #intDatoteka

  ' Določimo minimalno dolžino ukaznega polja
  ReDim Preserve lngOpisRisbe(intUkaz - 1)

  ' Shranimo enolično oznako datoteke
  Put #intDatoteka, 1, DatotecnaCarovnija
  ' Shranimo ukaze
  Put #intDatoteka, , lngOpisRisbe
  Close #intDatoteka

' Izklop obravnave izvajalnih napak
On Error GoTo 0

  ' Obnovimo velikost ukaznega polja
  ReDim Preserve lngOpisRisbe(constMaxUkazov)

  ' Označimo shranjene spremembe
  blnShranjeno = True
  strDatoteka = strImeDatoteke
  intRazveljavitve = 0

  ' Obnovi napis na oknu
  frmGlavni.Caption = App.Title & " - " & IzlusciDatoteko(strDatoteka)

  ShraniDatoteko = True
  Exit Function

NapakaPreklic:
  ShraniDatoteko = False
  Exit Function
```

```
NapakaDatoteka:
  Close #intDatoteka
  MsgBox "Med shranjevanjem je prišlo do naslednje napake:" & vbCrLf & _
    Err.Description, vbCritical
  OdpriDatoteko = False

End Function
```

Če datoteka še ni poimenovana, prikažemo pogovorno okno. Ime datoteke vedno preberemo iz parametra, tako da lahko isto funkcijo uporabimo za shranjevanje pod znanim in pod novim imenom. Značilnost funkcije sta dva različna prestrezovalnika napak. Ni ju težko napisati, paziti moramo le, da v prvem predčasno končamo izvajanje funkcije, da ne bi po pomoti padli še v drugega.

Stavek za odpiranje datoteke poznamo že iz Urejevalnika. Pri odpiranju *binarne* datoteke ima naslednjo splošno obliko:

```
Open pot For Binary _
[[Shared|Lock[ {Read|Write|Read Write}]] As [#]ročica
```

Pri binarnih datotekah imamo popolno svobodo, zato nam pri odpiranju ni treba določiti, ali bomo iz njih brali ali vanje pisali, ker lahko počnemo oboje. To bi lahko imelo hude posledice pri prepisovanju obstoječe datoteke. Če bi bil nov opis slike krajši, bi prepisali le njen začetni del. Taka datoteka bi ne opisovala več pravilne slike. Zato pred zapisom preverimo morebitni obstoj datoteke, v katero želimo pisati s funkcijo Dir, in jo po potrebi odstranimo. Če datoteke pri odpiranju še ni, se samodejno ustvari nova.

Zapis v binarno datoteko je mogoč s stavkom **Put**, katerega popolna oblika se glasi:

```
Put [#]ročica, [zaporednoMesto], spremenljivka
```

Stavek na podano zaporedno mesto zapiše vso vsebino spremenljivke. Zaporedna mesta se začnejo s številko 1. Če ga izpustimo, se privzame trenutni položaj znotraj datoteke.

Ker stavek Put zapiše vso vsebino spremenljivke, tudi niza ali polja, moramo pred zapisom dolžino polja prilagoditi dejansko porabljeni vsebini. Da ima datoteka res minimalno dolžino, ni pomembno toliko zaradi učinkovitejše izrabe diska. Dolžina datoteke je informacija, ki je temelj za poznejšo obnovitev risbe, kakor bomo spoznali, ko bomo obravnavali njeno odpiranje.

> **NAMIG** Zaradi širjenja in krčenja polja učinkovitost programa ne trpi, saj se pri krčenju s stavkom ReDim Preserve vsebina polja ne prepisuje, ker ne potrebujemo dodatnega prostora. Večjih zakasnitev ni tudi pri obnovitvi velikosti, saj je ob krčenju visual basic le označil novo dolžino polja, neporabljenega prostora pa najverjetneje še ni sprostil. Tako dobimo na koncu nazaj povsem isti kos pomnilnika kakor na začetku.

Odpiranje

Pri uporabi binarnih datotek je zapis vsebine zelo preprost. Tudi branje ni posebej zapleteno, vendar moramo prebrano informacijo logično ovrednotiti. Binarna datoteka namreč ni nič drugega kakor zaporedje zlogov. Kaj kateri pomeni, je v celoti odvisno od naše programske kode.

Tudi vsa opravila, povezana z odpiranjem datoteke, so zbrana v enem podprogramu:

```
' Odpiranje datoteke
Private Function OdpriDatoteko()
Dim intDatoteka As Integer
Dim strCarovnija As String

  With dlgDialog
    .DialogTitle = "Shrani risbo"
    .Flags = cdlOFNFileMustExist + cdlOFNHideReadOnly
    .filename = strDatoteka
    .CancelError = True

' Prvi prestreznik skrbi za morebitni preklic
On Error GoTo NapakaPreklic
    .ShowOpen
    strDatoteka = .filename
  End With

' Od tod naprej skrbimo za datotečne napake
On Error GoTo NapakaDatoteka
  intDatoteka = FreeFile

  ' Branje iz datoteke
  Open strDatoteka For Binary As #intDatoteka
  ' - preveri čarovnijo
  strCarovnija = DatotecnaCarovnija
  Get #intDatoteka, 1, strCarovnija
  If strCarovnija <> DatotecnaCarovnija Then
    Err.Raise vbObjectError + 1, , "Napačna oblika zapisa datoteke"
  End If
  ' - preberi datoteko v polje
  Get #intDatoteka, , lngOpisRisbe

  ' Določimo število ukazov
  intUkaz = ((LOF(intDatoteka) - Len(strCarovnija)) / 4)

  Close #intDatoteka

' Izklop obravnave izvajalnih napak
On Error GoTo 0

  ' Določimo stanje brez sprememb
  blnShranjeno = True

  ' Osvežimo napis okna
  frmGlavni.Caption = App.Title & " - " & IzlusciDatoteko(strDatoteka)

  ' Obnovimo risbo
  ObnoviRisbo
```

```
  OdpriDatoteko = True
  Exit Function

NapakaPreklic:
  OdpriDatoteko = False
  Exit Function

NapakaDatoteka:
  Close #intDatoteka
  MsgBox "Med odpiranjem je prišlo do naslednje napake:" & vbCrLf & _
    Err.Description, vbCritical
  OdpriDatoteko = False

End Function
```

Pri branju datoteko odpremo enako. Najprej poizkusimo iz nje prebrati datotečni podpis. Branje iz binarne datoteke je mogoče s stavkom **Get**. Njegova oblika je podobna stavku Put:

```
Get [#]ročica, [zaporedneMesto], spremenljivka
```

Stavek z zaporednega mesta, ki ga lahko izpustimo, poizkusi prebrati natančno toliko zlogov, kolikor jih lahko shrani podana spremenljivka. Če je datoteke prej konec, bo prebrana vsebina krajša, izvajalne napake pa zaradi tega ne bo.

Zaradi značilnosti stavka Get spremenljivko, v katero bomo prebrali podpis, najprej prilagodimo bodoči vsebini. Če se ta po branju ne ujema več s podpisom datoteke Risarja, *povzročimo* napako med izvajanjem. Natančnejši opis predmeta Err in metode Raise smo spoznali v prejšnjem poglavju. Napako, povzročeno z branjem iz napačne datoteke, prestreže drugi prestreznik, ki datoteko zapre in prikaže obvestilo.

Če je s podpisom vse v redu, sledi branje opisa slike. Ker ima polje z opisom slike vnaprej določeno velikost, po branju ne moremo vedeti, koliko zlogov smo dejansko prebrali. Tu naletimo na osnovno težavo pri rabi binarnih datotek: ovrednotenje prebrane vsebine. Mi imamo na voljo dvoje:

- Posodobimo podprogram za zapisovanje. Ta naj pred opisom risbe v binarno datoteko zapiše še število pozneje zapisanih ukazov. Pri branju bomo najprej prebrali to vrednost in izvedeli, koliko ukazov bo sledilo.
- Število shranjenih ukazov izračunamo iz dolžine datoteke.

Izbrali smo drugo možnost, ker je za nas preprostejša. Če bi binarna datoteka vključevala podatke različnih vrst, bi bila ta možnost precej zapletena, če ne celo nemogoča.

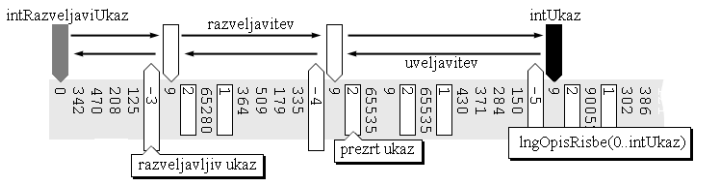
Ob koncu podprograma z novo, pravkar prebrano vsebino obnovimo risbo in uskladimo napis na oknu z imenom datoteke. Med delom smo uporabili uslužnostno funkcijo IzlusciDatoteko, razvito že pri Okenskem komandantu.

Razveljavitev in uveljavitev

Uporaba posebne podatkovne strukture, ki hrani vsak ukaz risbe, nam omogoča razviti podprograme za razveljavitev in uveljavitev poljubnega števila ukazov. Pri določanju konstantnih vrednosti, ki opisujejo ukaze, smo določene ukaze ponazorili s pozitivnimi, večino pa z negativnimi števili. Tako označevanje nam omogoča preprost sprehod po polju ukazov. Ko želimo kak ukaz razveljaviti, se enostavno sprehodimo nazaj do prejšnjega negativnega števila. Ta zagotovo predstavlja ukaz, saj se negativne koordinate, negativna debelina ali barva, predstavljena z negativnim številom, v našem opisu risbe ne morejo pojaviti.

Pri razveljavitvi moramo poskrbeti le še za to, da si zapomnimo kazalec zadnjega izvršenega ukaza. Ta kazalec predstavlja mejo, do katere lahko razveljavljene ukaze vnovič uveljavljamo. Če uporabnik razveljavi nekaj ukazov, zatem pa izriše nov element, se mora ta kazalec ustrezno vnovič nastaviti, drugače bi se lahko zgodilo kaj nepredvidljivega.

Shematsko je postopek razveljavljanja in uveljavljanja prikazan na spodnji sliki:



Slika 10-33: Ponazoritev razveljavljanja in uveljavljanja ukazov

Ustrezna programska koda:

```
' Razveljavitev ukaza
Private Sub RazveljaviUkaz()
Dim lngShraniBarvo As Long
Dim intShraniDebelino As Integer
  ' Shrani nastavitve
  lngShraniBarvo = lngBarva
  intShraniDebelino = intDebelina

  ' 4 osnovni ukazi začnejo sliko
  If intUkaz = 4 Then Exit Sub

  ' Najdemo prejšnji ukaz
  intUkaz = intUkaz - 1
  If intUkaz > intRazveljavitve Then intRazveljavitve = intUkaz
  Do While (intUkaz > 4 And lngOpisRisbe(intUkaz) >= 0)
    intUkaz = intUkaz - 1
  Loop
  ' In obnovimo risbo
  ObnoviRisbo

  ' Uveljavi izbrane nastavitve
  lngBarva = lngShraniBarvo
  intDebelina = intShraniDebelino
End Sub
```

```
' Uveljavitev ukaza
Private Sub UveljaviUkaz()
Dim lngShraniBarvo As Long
Dim intShraniDebelino As Integer
  ' Shrani nastavitve
  lngShraniBarvo = lngBarva
  intShraniDebelino = intDebelina

  ' Preverimo mejo uveljavljanja regularnih ukazov
  If intRazveljavitve = 0 Or intUkaz > intRazveljavitve Then Exit Sub

  ' Najdemo naslednji ukaz
  intUkaz = intUkaz + 1
  Do While (intUkaz <= intRazveljavitve And lngOpisRisbe(intUkaz) >= 0)
    intUkaz = intUkaz + 1
  Loop
  ' In obnovimo risbo
  ObnoviRisbo

  ' Uveljavi izbrane nastavitve
  lngBarva = lngShraniBarvo
  intDebelina = intShraniDebelino
End Sub
```

V obeh podprogramih moramo paziti, da obnovimo nastavitve barve in debeline, saj po razveljavitvi/uveljavitvi in vnovičnem obnavljanju risbe morda ne veljajo več. Ne pozabimo na zapis odzivnih podprogamov za ustrezna menujska ukaza:

```
' Menujski ukazi
Private Sub mnuUrediRazveljavi_Click()
  RazveljaviUkaz
End Sub

Private Sub mnuUrediUveljavi_Click()
  UveljaviUkaz
End Sub
```

Prenos na odložišče

Skupaj z risanjem likov smo omogočili tudi izris označevalca (*marquee*), ki ga animiramo z uro. Območje, ki ga označuje, bomo uporabili pri prenosu na odložišče. Za prenos slike iz grafičnega vsebnika imamo na voljo metodo SetData vgrajenega predmeta Clipboard. Vendar ima ta metoda dve pomanjkljivosti:

- Prenese lahko le vso sliko v gradniku.
- Prenaša samo obstojno grafiko.

Za nas tako vedenje ne pride v poštev. Radi bi prenesli poljubno območje grafike, ki je izrisana z grafičnimi metodami.

Pomožni gradnik

Pri obravnavanju grafike smo spoznali uslužnostno metodo za prepisovanje poljubnega pravokotnega območja, imenovano PaintPicture. Ta metoda je na voljo za grafične vsebnike. Edini, ki ga zaenkrat še ne uporabljamo, je obrazec. Žal ni uporaben zaradi omejitev metode SetData. Izkaže se, da lahko težavo rešimo samo z uporabo novega, nevidnega slikovnega okvira, katerega velikost bomo dinamično spreminjali glede na izbrano območje.

Na obrazec moramo na poljubno mesto dodati nov gradnik slikovniOkvir.

Gradnik	Lastnost	Nastavitev
slikovni okvir (*PictureBox*)	(Name)	picClipCopy

Druge vzpostavitve bomo opravili v malce popravljeni proceduri dogodka Load (prikazan je le spremenjeni del):

```
' Začetne nastavitve ob nalaganju obrazca
Private Sub Form_Load()

( ... )

  ' - slikovni okvir za pomoč pri kopiranju
  With picClipCopy
    .AutoRedraw = True
    .Visible = False
    .ScaleMode = 3
  End With

( ... )

End Sub
```

Slikovni okvir mora imeti nastavljeno lastnost AutoRedraw. S tem bo vsa – izrisana in prenesena – grafika v njem postala obstojna in kot taka dostopna metodi SetData. Oglejmo si prvi poizkus pomožne procedure za prenos slike:

```
' Prenos risbe na odložišče
Private Sub KopirajRisbo()

  If udtClip.P1.x = -1 Then
    udtClip.P1.x = 0
    udtClip.P1.y = 0
    udtClip.P1.x = picRisba.Width
    udtClip.P1.y = picRisba.Height
  Else
    ObnoviRisbo
  End If

  With picClipCopy
    .Width = Abs(udtClip.P2.x - udtClip.P1.x)
    .Height = Abs(udtClip.P2.y - udtClip.P1.y)
    .Cls
```

```
    .PaintPicture picRisba.Image, _
      0, 0, .Width, .Height, _
      Min(udtClip.P1.X, udtClip.P2.X), _
      Min(udtClip.P1.Y, udtClip.P2.Y),_
      .Width, .Height, vbSrcCopy
    Clipboard.Clear
    Clipboard.SetData .Image
    .Cls
    .Width = 0
    .Height = 0
  End With
  udtClip.P1.x = -1

End Sub
```

V proceduri najprej preverimo obseg označenega območja. Če je nadzorovana vrednost ostala na nemogoči koordinati –1, bomo uporabili vso risbo. V nasprotnem primeru je izrisan označevalec, ki se ga znebimo z obnovitvijo risbe. Za uspešen prenos samo izbranega območja najprej ustrezno nastavimo velikost sicer nevidnega gradnika picClipCopy. Po popravku velikosti vanj prenesemo označeni izsek slike z metodo PaintPicture. Prenos na odložišče začne metoda Clear, ki odstrani morebitno vsebino, končni prenos pa opravi metoda SetData. Kot parameter smo ji posredovali lastnost Image in ne Picture. Lastnost Image opisuje obstojno izrisano grafiko, njena uporaba je smiselna le, če ima gradnik postavljeno lastnost AutoRedraw (za to smo poskrbeli ob vzpostavitvi gradnika picClipCopy). Lastnost Picture predstavlja obstojno sliko, prebrano iz zunanje datoteke. Na koncu moramo sprostiti pomnilnik, ki ga zaseda nevidna kopija risbe. Ta je lahko velika kot osnovna slika, zato so njene pomnilniške zahteve lahko velikanske. Zbrišemo vsebino gradnika in ga fizično zmanjšamo na minimum. Na koncu znova označimo neobstoječe območje označitve.

V proceduri uporabljamo pomožno funkcijo Min, ki vrne manjše od dveh posredovanih števil. Ta funkcija v visual basic ni vgrajena, ampak smo jo razvili posebej. Ker je splošno uporabna, jo uvrstimo v posebni modul s kodo:

```
' Vrne manjše od posredovanih števil
Public Function Min(ByVal a As Integer, ByVal b As Integer) As Integer
  If a > b Then
    Min = b
  Else
    Min = a
  End If
End Function
```

V glavnem obrazcu moramo poskrbeti za uporabo podprograma kopiranja z odzivom na pripadajoči menujski ukaz:

```
Private Sub mnuUrediKopiraj_Click()
  KopirajRisbo
End Sub
```

Za preizkus lahko ob vzpostavitvi začasno izključimo stavek za nastavitev nevidnosti, ki ga nadomestimo s klicem metode ZOrder. Ta bo postavila gradnik povsem v ospredje, tako da lahko njegovo vedenje opazujemo. Žal nas čaka bridko razočaranje. Tudi metoda PaintPicture deluje le z obstojno grafiko. Uporabna je – podobno kakor metoda SetData – le pri grafičnih vsebnikih z vključeno lastnostjo AutoRedraw. To pri nas enostavno ne pride v poštev, pomnilniške zahteve programa bi pri večjih zaslonskih ločljivostih postale enostavno prevelike.

Na pomoč, API!

Česar ne zmorejo metode visual basica, ponavadi zmore katera izmed funkcij API. Že pri obravnavanju metode PaintPicture smo omenili, da je bilo pred njenim nastankom treba prenašati izrezana območja slik s posebej hitro zunanjo funkcijo. Njeno ime je BitBlt. S pripomočkom Text API Viewer dobimo njej ustrezen stavek Declare, ki ga zapišemo v del splošnih napovedi obrazca (zapisan je le delno):

```
' Zunanja funkcija za pomoč pri kopiranju
Private Declare Function BitBlt Lib "gdi32" (ByVal hDestDC As Long, …
```

Funkcija za hiter prepis pravokotnih območij potrebuje ročico risalne površine, ki je programom v visual basicu dostopna prek lastnosti hDC grafičnih vsebinikov. To lastnost smo omenili, ko smo obravnavali gradnik ListImage, ki pri svoji metodi Draw prav tako zahteva to lastnost. Ročica risalne površine predstavlja vso grafiko, obstojno in tisto, izrisano z grafičnimi metodami, zato bo funkcija BitBlt rešila našo težavo. Njeni parametri so nadvse podobni parametrom metode PaintPicture, kar nam da slutiti, da je ta metoda le funkcija BitBlt v preobleki. Žal v napačni.

V proceduri za prenašanje slike zgolj zamenjamo klic metode PaintPicture s stavkom:

```
Private Sub KopirajRisbo()

( . . . )

    BitBlt .hDC, 0, 0, .Width, .Height, _
      picRisba.hDC, _
      Min(udtClip.P1.x, udtClip.P2.x), _
      Min(udtClip.P1.y, udtClip.P2.y), vbSrcCopy

( . . . )

End Sub
```

Zgled zdaj deluje, zato lahko gradnik picClipCopy znova spravimo v nevidno stanje.

> **NAMIG** PaintPicture je pravzaprav preoblečena funkcija API po imenu StretchBlt, ki za razliko od BitBlt omogoča tudi razteg ali krčenje pri prepisovanju območja. S funkcijo API bi bilo mogoče zaobiti tudi drugo težavno metodo, metodo SetData. Žal je nadzor odložišča samo s funkcijami API nekoliko zapleten, zato je dela kar precej.

Izvoz bitne slike

Zelo podobne težave, kakor smo jih imeli pri prenašanju izreza risbe na odložišče, bi imeli tudi pri poizkusu njenega shranjevanja v bitno sliko. Z zgoraj uporabljeno tehniko pomožnega gradnika in funkcije API lahko rešimo tudi izvoz risbe v datoteko z bitno sliko .BMP. Ustrezna procedura je:

```
' Izvoz bitne slike
Private Function IzvoziKotBitnoSliko() As Boolean
Dim strStariFilter As String
Dim strStariExt As String
Dim strImeDatoteke As String

  ' Prilagodimo in uporabimo pogovorno okno
  With dlgDialog
    .DialogTitle = "Izvozi risbo kot bitno sliko"
    .Flags = cdlOFNOverwritePrompt
    ' - shranimo nastavitve
    strStariFilter = .Filter
    strStariExt = .DefaultExt
    ' - določimo nov filter in podaljšek
    .Filter = "Bitne slike (*.bmp)|*.bmp|Vse datoteke (*.*)|*.*"
    .DefaultExt = ".bmp"
    ' - določimo začetno vsebino
    If strDatoteka <> "" Then
      .filename = ZamenjajPodaljsek(strDatoteka, ".bmp")
    Else
      .filename = ""
    End If
    .CancelError = True

On Error GoTo Napaka
    .ShowSave
    ' - obnovimo stare nastavitve
    .Filter = strStariFilter
    .DefaultExt = strStariExt
    strImeDatoteke = .filename
  End With

  ' Preprečimo shranjevanje pokvarjene (neobnovljene) risbe
  ObnoviRisbo

  ' Uporabimo pomožno sliko
  With picClipCopy
    ' - prenesemo celotno risbo
    .Width = picRisba.Width
    .Height = picRisba.Height
    .Cls
    BitBlt .hDC, 0, 0, .Width, .Height, picRisba.hDC, 0, 0, vbSrcCopy
    ' - shranimo bitno sliko
    SavePicture picClipCopy.Image, strImeDatoteke
    ' - sprostimo pomnilnik
    .Cls
    .Width = 0
    .Height = 0
  End With
```

```
  IzvoziKotBitnoSliko = True
  Exit Function

Napaka:
  ' Tudi ob preklicu moramo obnoviti stanje
  dlgDialog.Filter = strStariFilter
  dlgDialog.DefaultExt = strStariExt
  IzvoziKotBitnoSliko = False
End Function
```

Nekaj več dela imamo zaradi začasne prilagodtve standardnega pogovornega okna, ki mora v tem primeru kazati na datoteke s podaljškom .BMP. V ta namen smo razvili še posebno uslužnostno funkcijo, ki zamenja podaljšek poljubne datoteke. Najbolje, da jo uvrstimo v ločeni modul s kodo:

```
' Spremeni podaljšek končnega elementa poti
Public Function ZamenjajPodaljsek(ByVal strNiz As String, _
ByVal strPodaljsek As String) As String
Dim i, j As Integer
  i = 0
  Do
    j = i
    i = InStr(i + 1, strNiz, ".")
  Loop While i > 0
  If j > 0 Then
    ZamenjajPodaljsek = Left(strNiz, j - 1) & strPodaljsek
  Else
    ZamenjajPodaljsek = strNiz & strPodaljsek
  End If
End Function
```

Dejansko shranjevanje opravi ena sama vgrajena funkcija, **SavePicture**. Ta shrani celotno obstojno grafično vsebino podanega vsebnika v obliki datoteke BMP.

> **NAMIG** Metoda LoadFile uspešno prebere tudi datoteke GIF in JPEG. Žal shranjevanje s funkcijo SavePicture ni tako splošno.

Priveski

Risar je skoraj končan. Manjka mu še sprotna pomoč, ki jo bomo pripravili v nadaljevanju. Prej si oglejmo, kako lahko program opremimo z vizitko in pozdravnim oknom.

Vizitka

Z vizitko pravzaprav nimamo nobenega dela. Uporabimo le predlogo, ki smo jo pripravili pri razvoju Urejevalnika. Ker je zapisana v povsem prenosljivi kodi, jo lahko brez popravkov uporabimo. Dodati moramo le ustrezen menujski ukaz. Po želji bi lahko vizitko opremili še z izpisom nekaj statističnih podatkov, kot je npr.

trenutno izbrana barva, debelina in število pomnjenih ukazov. To bomo postorili grafično, v odzivu na dogodek Paint.

V menuju **Project** izberemo ukaz **Add Form**... in v galeriji izberemo pripravljeno predlogo. Tako dobljeno vizitko lahko prikažemo z odzivom na ustrezen ukaz menuja Pomoč. S projektnim oknom prikažemo vmesnik glavnega okna, izberemo ustrezen menujski ukaz ter zapišemo vrstico za prikaz vizitke:

```
Private Sub mnuPomocVizitka_Click()
  frmVizitka.Show vbModal, Me
End Sub
```

Preveriti moramo, ali je ime obrazca z vizitko res frmVizitka. Povzame se po pripravljeni predlogi. S tem smo lahko že povsem zadovoljni, lahko pa vizitko še izpilimo.

Če želimo znotraj vizitke izpisati določene podatke o osnovnem obrazcu, moramo njenim podprogramom omogočiti dostop do njih. V našem zgledu to najlaže storimo tako, da ustrezne spremenljivke označimo kot javne. V delu splošnih napovedi glavnega obrazca napravimo naslednje popravke:

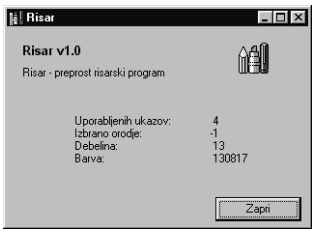
```
Public lngBarva As Long
Public intDebelina As Integer
Public intLik As Integer
( ... )
Public intUkaz As Integer
```

Znotraj programske kode vizitke pa zapišimo vsebino naslednje odzivne procedure:

```
Private Sub Form_Paint()
  ScaleMode = vbPixels
  CurrentX = lblOpis.Left
  CurrentY = lblOpis.Top + 50
  Print Tab; "Uporabljenih ukazov: ", frmGlavni.intUkaz
  Print Tab; "Izbrano orodje: ", frmGlavni.intLik
  Print Tab; "Debelina: ", , frmGlavni.intDebelina
  Print Tab; "Barva: ", , frmGlavni.lngBarva
End Sub
```

Videz vizitke si oglejte na naslednji sliki.

Slika 10-34:
Vizitka Risarja

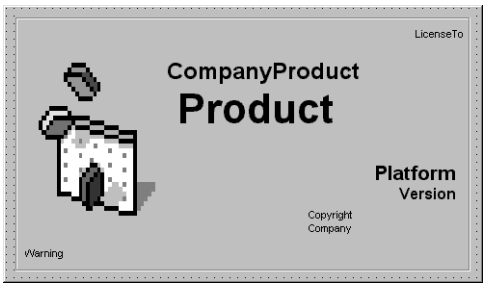


Pozdravno okno

Precej programov nas ob nalaganju pozdravi z oknom, ki ponudi osnovne podatke o programu. Včasih ta okna, v žargonu imenovana *splash screen*, ponudijo še natančnejši vpogled v postopek vzpostavljanja. Risar se ne vzpostavlja toliko časa, da bi bil poseben uvodni zaslon potreben. Pripravili ga bomo le zato, da si ogledamo način njegove izdelave in spoznamo nekaj uporabnih zmožnosti ponazarjanja vzpostavitvenih opravil.

Pri izdelavi pozdravnega okna nam lahko pomaga že pripravljena predloga. Obrazec, ki temelji na njej, dodamo v projekt z ukazom **Project ➪ Add Form**... Izmed pripravljenih predlog izberemo »Splash Screen«. Videz tako dobljenega obrazca si oglejte na spodnji sliki:

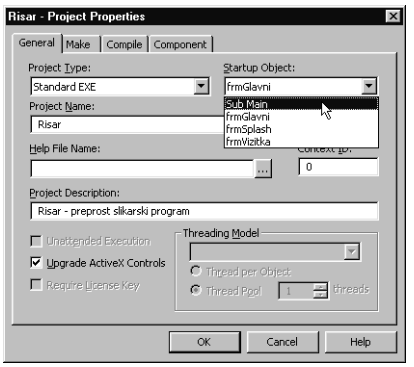
Slika 10-35: Predloga za pozdravno okno



Prvi približek uvodnega zaslona tako že imamo, sedaj moramo poskrbeti za njegov prikaz. Uspešen prikaz uvodnega zaslona bo mogoč samo, če začetek izvajanja programa prestavimo v samostojni modul s kodo, znotraj posebne procedure Sub Main, ki bo nadzorovala začetni prikaz obrazcev. Pri vsakem projektu lahko določimo, da se njegovo izvajanje začne v enem izmed pripravljenih obrazcev ali pa v posebni proceduri Sub Main, ki bo prevzela odgovornost za vzpostavitev projekta.

Ustrezno nastavitev določimo z ukazom **Project ➪ Risar properties**... (ime ukaza je odvisno od naziva projekta). V pogovornem oknu za nastavitev lastnosti projekta najdemo pod jezičkom **General** zaprti seznam **Startup Object**. V njem so vedno našteti vsi razpoložljivi obrazci in omenjena posebna procedura. Izberimo torej Sub Main.

Slika 10-36: Izbor začetka izvajanja projekta



Procedura Sub Main je lahko le v neodvisnem modulu s kodo. Mora biti procedura, funkcija Main se ne bi upoštevala. Modul s kodo v našem projektu že imamo, ker je v njem nekaj pomožnih funkcij. Dodajmo mu še kodo za prikaz uvodnega zaslona:

```
Sub Main()
  frmSplash.Show
  frmGlavni.Show
End Sub
```

Če projekt preizkusimo, bomo opazili dve težavi:

- Uvodno okno je vidno le kratek čas.
- Ob koncu dela moramo ročno zapreti še uvodno okno, drugače se izvajanje programa ne konča.

Vsi projekti se izvajajo, dokler je vidno vsaj še eno okno, če pa se njihovo izvajanje začne v proceduri Main, se mora poleg vseh oken končati tudi to. Novi poizkus je lahko tak:

```
Sub Main()
  frmSplash.Show
  frmGlavni.Show
  Unload frmSplash
End Sub
```

S tem sicer rešimo težavo z zapiranjem, vendar je uvodno okno še vedno skorajda neopazno, ker ga glavni obrazec zelo hitro prekrije. Na splošno je lahko uvodno vzpostavljanje glavnega obrazca precej dolgotrajnejše. Resnici na ljubo, uvodno okno je pravzaprav potrebno zgolj takim projektom. Če uporabnika programa posiljujemo z uvodnim oknom, ki je samemu sebi namen, si bomo naredili medvedjo uslugo. Dolgotrajna obdelava je lahko tudi sestavni del zagonske procedure. Oglejmo si tak zgled:

```
Sub Main()
Dim i As Integer
  frmSplash.Show
  For i = 0 To Screen.FontCount
    frmSplash.lblWarning.Caption = Screen.Fonts(i)
    DoEvents
  Next
  frmGlavni.Show
  Unload frmSplash
End Sub
```

V zgornjem zgledu ponazarjamo dolgotrajno operacijo s tem, da v pozdravnem oknu izpisujemo razpoložljive zaslonske pisave. V čem je pomen funkcije **DoEvents**? Ta funkcija prepusti procesor operacijskemu sistemu, ki ima tako čas urediti čakajoča opravila. Eno izmed njih je tudi dokončen prikaz vsebine oznake lblWarning. Če bi funkcije ne poklicali, bi se oznaka sploh ne osvežila. Vsiljeno osvežitev lahko dosežemo tudi s klicem metode Refresh, toda v tem primeru bi se obrazec ne izrisal do konca. No, tudi tega lahko prisilno osvežimo enako.

NAMIG Pomen funkcije DoEvents je bil mnogo bolj izrazit v 16-bitnih različicah Oken, ki niso poznala predkupne večopravilnosti, ampak je bil vsak posamezen progam zadolžen za to, da je med intenzivnim delom tu in tam za kratek čas prepustil nadzor operacijskemu sistemu in drugim programom.

No, tudi Risarju bi se podalo pozdravno okno, čeprav ne pozna dolgotrajnih vzpostavitvenih opravil. To okno pa mora biti posebne vrste. Vztrajati mora na vrhu tudi po tem, ko se prikaže glavno okno. Preden se lotimo priprave posebnega pozdravnega okna, obnovimo zagonsko proceduro v naslednjo obliko:

```
Sub Main()
  frmSplash.Show
  frmSplash.Refresh
  frmGlavni.Show
End Sub
```

Sedaj se posvetimo prilagoditvi pozdravnega okna. Da bo okno vztrajalo na vrhu, moramo uporabiti klic posebne funkcije API z imenom SetWindowPos. Njeno napoved in vrednosti nekaterih konstant, ki jih bomo potrebovali pri klicu, najdemo z orodjem Text API Viewer. Če bo okno vztrajalo na vrhu, bomo morali poskrbeti za njegovo odstranitev. Pripravljena predloga sicer vključuje kodo, ki obrazec odstrani, ko ga kliknemo ali pritisnemo na tipko, vendar moramo vgraditi zanesljivejši mehanizem. Tu nam priskoči na pomoč gradnik ura (*Timer*), ki ga dodamo uporabniškemu vmesniku pozdravnega okna.

Gradnik	Lastnost	Nastavitev
ura (*Timer*)	(Name)	Timer1
	Interval	100
	Enabled	True

Videz kode pozdravnega okna je razviden iz spodnjega izpisa. S predlogo pripravljene odzivne metode za odstranjevanje obrazca smo odstranili, dodali pa smo odstranitev pri dogodku Timer:

```
Option Explicit
Private Declare Function SetWindowPos ... ' manjka
Const intMaxTick = 15

Private Sub Form_Load()
Const HWND_TOPMOST = -1
Const SWP_NOMOVE = &H2
Const SWP_NOSIZE = &H1
  SetWindowPos Me.hwnd, HWND_TOPMOST, 0, 0, 0, 0, SWP_NOMOVE + SWP_NOSIZE
  lblVersion.Caption = "Version " & App.Major & "." & _
    App.Minor & "." & App.Revision
  lblProductName.Caption = App.Title
End Sub
```

```
Private Sub Timer1_Timer()
Static intTick As Integer
  intTick = intTick + 1
  If intTick = intMaxTick Then Unload Me
End Sub
```

V odzivu na dogodek Timer povečujemo statično spremenljivko. Šele ko ta doseže vnaprej predpisano vrednost, obrazec tudi dokončno odstranimo. Seveda bi lahko že takoj določili daljši interval ure, vendar nam tak pristop omogoča, da med prikazom pozdravnega okna še kaj postorimo, npr. razveselimo uporabnika z zanimivim grafičnim učinkom. Tu je le zamisel:

```
Private Sub Timer1_Timer()
Static intTick As Integer
  intTick = intTick + 1
  DrawWidth = intTick
  Line (Rnd * ScaleWidth, Rnd * ScaleHeight)-(Rnd * ScaleWidth, _
    Rnd * ScaleHeight), RGB(Rnd * 256, Rnd * 256, Rnd * 256)
  If intTick = intMaxTick Then Unload Me
End Sub
```

Zanimivo, da lahko tudi med prikazom pozdravnega zaslona že uporabljamo glavni obrazec. Koda obeh se namreč izvaja sočasno.

NAMIG Risar, razvit do te točke, najdete na priloženi plošči CD-ROM v mapi \Primeri\2-Delavnica\3-Risar\3-Koda\.

Sprotna pomoč

Najsi je program še tako preprost, če ga želimo ponuditi drugim uporabnikom, mu moramo priložiti vsaj osnovna navodila. Lepo je, če so elektronska. Okna ponujajo vsem programom uporabo sistemskega mehanizma za prikaz sprotne pomoči. Ta mehanizem je za vse programe enak, zato ga večina uporabnikov že pozna. Z Okni 95 so sistem sprotne pomoči še izboljšali. Vsem gradnikom lahko prepišemo kratke namige (*tooltips*) in daljše opise, ki jih sistem prikazuje v posebnih okoliščinah. Preden lahko sistemske storitve uporabimo, moramo pripraviti ustrezna navodila.

Razvoj datoteke s pomočjo

Za razvoj datoteke s pomočjo potrebujemo urejevalnik besedila, ki zna zapisovati besedilo v obliki *rich text* (RTF) in podpira posebne načine označevanja besedila v datotekah s pomočjo. Praktično noben sodobni urejevalnik besedil s tem nima težav, žal pa v ta namen ni uporaben pripomoček WordPad, ki je priložen Oknom. Resda zna uporabljati zapis RTF, vendar ne podpira vseh potrebnih označevanj.

Pripravljeno besedilo moramo – podobno kakor pripravljena sredstva – prevesti z zunanjim prevajalnikom, *Help Compiler*. Na srečo smo za razvoj pomoči znatno bolje opremljeni kakor pri delu s sredstvi. Na plošči CD-ROM boljših različic visual basica v mapi \Tools\Hcw\ najdemo orodje Microsoft Help Workshop, ki je čistokrvni okenski program za končno pripravo sprotne pomoči. Preden ga uporabimo, ga moramo namestiti.

> **NAMIG** Za razvoj pomoči je na voljo precej orodij neodvisnih izdelovalcev.

Priprava besedila

Za pripravo besedila potrebujemo sodobnejši urejevalnik besedila, kot so Microsoft Word, Corel WordPerfect ali Lotus WordPro. Besedilo bodoče datoteke s pomočjo sestavimo iz več *tem*. Tema (*help topic*) je posamezna prikazana stran, pojasnilo ali namig. Pri njenem vnosu moramo spoštovati naslednja pravila:

- Vsaka tema je zapisana na lastni strani. Med njimi mora biti postavljen trdi prelom strani (*hard page break*).
- Vsaka tema mora imeti enolično, neponovljivo *oznako* (*context string*), ki jo zapišemo kot niz znakov brez presledkov ali ločil.
- Vsaki temi lahko predpišemo naslov (*title*) in ključne besede (*keywords*), po katerih bo mogoče temo najti v stvarnem kazalu.
- Temam lahko določimo zaporedje pregledovanja (*browsing*) in medsebojne povezave (*hyperlinks*).

V besedilu lahko uporabljamo različno oblikovanje znakov, alinej in odstavkov. Večina oblikovnih navodil bo ohranjenih, nekoliko moramo paziti le pri izbiranju pisave, saj bo datoteka s pomočjo lahko v drugih računalnikih slabše berljiva. Najbolje je uporabiti pisavo »Times New Roman« oziroma »Arial«.

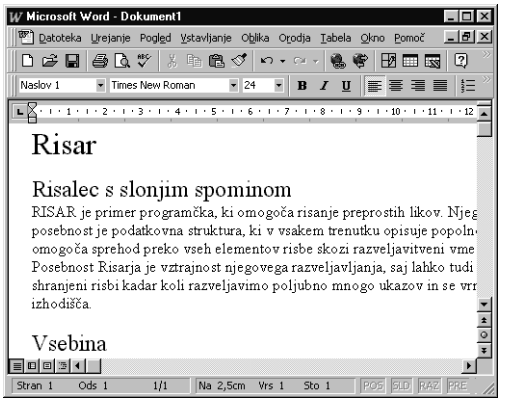
Večino posebnih navodil vnesemo v obliki sprotnih opomb (*footnotes*), ki jih označimo s posebnimi simboli. Ti simboli so zbrani v naslednji tabeli:

Simbol	Označuje
#	Oznako teme (*context string*).
$	Naslov teme (*title*).
K	Seznam ključnih besed (*keyword list*).
+	Zaporedje pregledovanja (*browsing*).

Vnos sprotnih opomb je odvisen od izbranega urejevalnika. Če uporabljamo Word, določimo sprotno opombo z ukazom **Vstavljanje ➪ Sprotne opombe**... (**Insert ➪ Footnote**...). Prikaže se pogovorno okno, v katerem določimo njeno oznako. Vnesemo jo v polje **Osebna oznaka** (**Custom mark**).

Vnesimo prvo temo sprotne pomoči Risarja. To naj bo kratek opis programa in kazalo s povezavami z drugimi stranmi. Najprej vnesemo besedilo. Naslov zapišemo v prvi vrstici s pisavo velikosti okrog 24 pik. Za preostalo besedilo uporabimo velikost 12 pik, mednaslove pa poudarimo s črkami vmesnih velikosti. Na koncu vstavimo prelom strani (v Wordu pritisnemo na kombinacijo tipk Ctrl+Enter).

Slika 10-37: Vnos besedila je prvi korak do datoteke s pomočjo



Oznako teme vnesemo tako, da se postavimo pred naslov, kazalec vnosne točke naj bo povsem na začetku besedila, desno od prvega znaka naslova. Vstavimo sprotno opombo, označeno s simbolom #, in vnesemo enolično oznako teme, npr.:

```
Vsebina
```

Določimo še naslov teme (opomba z oznako $), znova:

```
Vsebina
```

ter ključne besede (opomba z oznako K), npr.

```
vsebina;Risar;opis;posebnosti
```

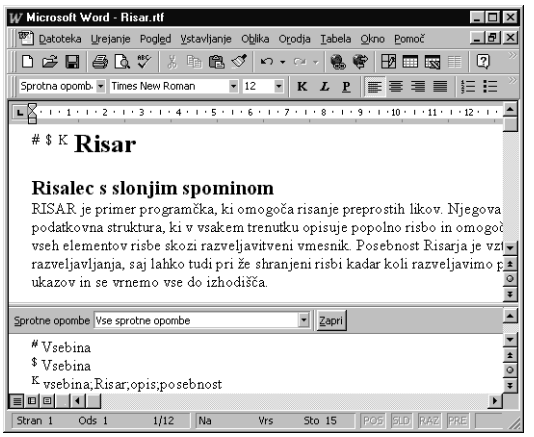
Več ključnih besed določimo tako, da jih razmejimo s podpičjem. Če želimo ključne besede urediti hierarhično, uporabimo vejico, npr.:

```
Risar,vsebina;Risar,opis;Risar,posebnosti
```

Vse naštete ključne besede bodo uvrščene pod pojem Risar.

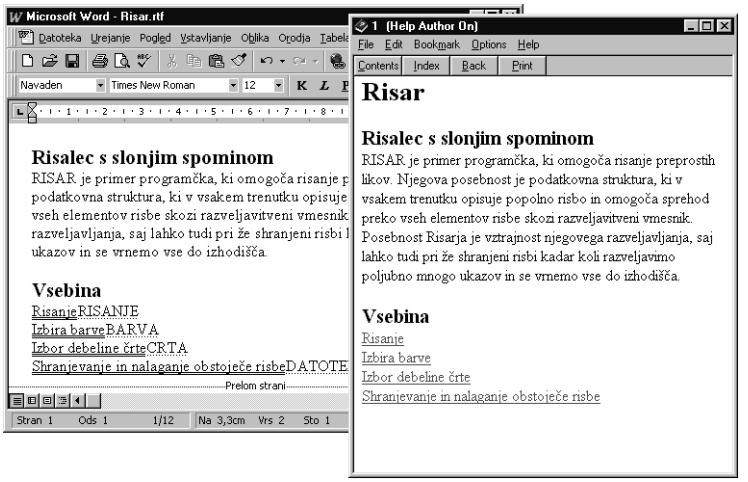
Sedaj pridejo na vrsto povezave z drugimi temami, ki jih bomo določili. Povezavo določimo tako, da izpostavljene besede dvakrat podčrtamo (*double underline*) ali

Slika 10-38: Z vnosom oznak dokončno opredelimo posamezno temo.



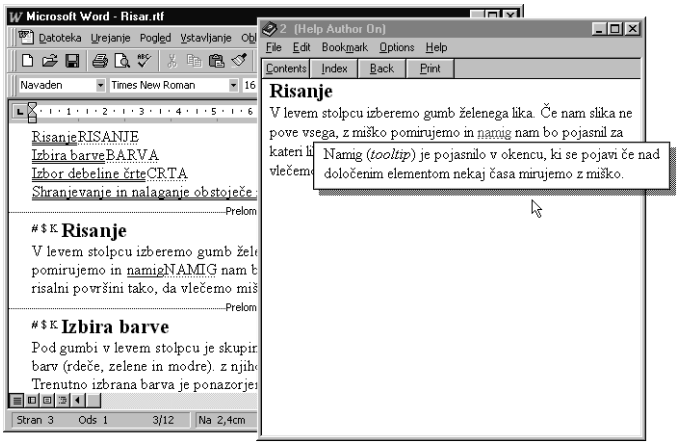
prečrtamo (*strikethrough*), tik poleg (brez vmesnih presledkov) pa vnesemo enolično oznako povezane teme. Vneseno oznako teme moramo označiti kot skrito besedilo (*hidden text*).

Slika 10-39: Označitev in videz povezave



NAMIG Če uporabljamo Word, lahko dvakratno podčrtanje in skrito besedilo določimo tako, da označimo želeni del besedila in izberemo ukaz **Oblika** ➪ **Pisava**... (**Format** ➪ **Font**...). Ko določimo skrito besedilo, to postane nevidno, kar pri izdelovanju pomoči ni preveč dobro. Prikaz skritega besedila omogočimo z vklopom stikala **Skrito besedilo** pod jezičkom **Pogled** ukaza **Možnosti**... iz glavnega menuja **Orodja**. (**Tools** ➪ **Options** ➪ jeziček **View** ➪ stikalo **Hidden Text**).

Posebna oblika povezav so *definicije*. Definicija (*definition*) je povezava, ki uporabnika pomoči ne pripelje v novo okno, temveč mu v manjšem oknu prikaže pojasnilo opazovanega pojma. Definicijo določimo tako, da izpostavljene besede označimo z enkratnim podčrtanjem, tik zraven pa kot skrito besedilo zapišemo oznako teme z njihovim pojasnilom.



Slika 10-40: Izpostavljene besede, potrebne dodatnega pojasnila, enkrat podčrtamo. Desno vidimo poznejši videz pojasnila.

Pripravljeno datoteko shranimo v obliki *rich text* pod imenom »Risalec.rtf«.

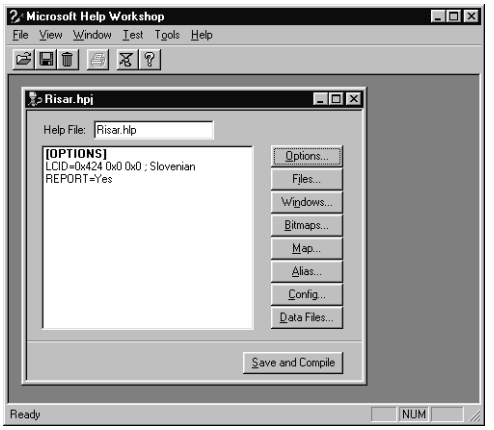
> **NAMIG** V Wordu izberemo ukaz **Datoteka ➪ Shrani kot**... ter s seznama **Shrani kot vrsto** izberemo **Obogateno besedilo** (**File ➪ Save As**..., seznam **Save As Type**, izberemo **Rich Text Format**).
>
> Pripravljeno datoteko z besedilom pomoči v obliki RTF najdete na priloženi plošči CD-ROM v mapi \Primeri\2-Delavnica\3-Risar\4-Pomoč\Help\.

Prevajanje pomoči

Za končno prevajanje pomoči uporabimo pripomoček Help Workshop. Omogoča nam pripravo projektne datoteke .HPJ (*Help Project*) in datoteke z opisom vsebine .CNT (*Help Content File*). Z njima in pripravljenim besedilom v datoteki RTF ustvari datoteko s sprotno pomočjo v datoteki .HLP. Uh, podaljškov, kolikor hočete.

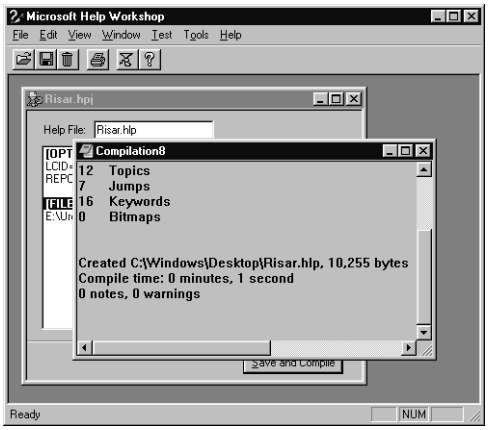
Najprej poženimo orodje Help Workshop. V menuju **File** izberemo ukaz **New** ter potrdimo možnost »Help Project«. Takoj moramo določiti ime datoteke – izberimo ime »Risar«. Pozdravi nas okno, v katerem bomo nastavili vse značilnosti projekta.

Slika 10-41: Projektno okno nastajajoče pomoči



Z gumbom **Options** prikažemo pogovorno okno osnovnih nastavitev. Zaenkrat jih še ne bomo spreminjali. Naslednji gumb, **Files**, odpre okno za določitev datotek z besedilom. Kliknemo gumb **Add**... ter poiščemo datoteko »Risalec.rtf«. Sedaj že lahko kliknemo gumb **Save and Compile** in prevajalnik nam bo poizkusil izdelati datoteko s pomočjo. O končnem stanju nas obvesti z izpisanim poročilom.

Slika 10-42: Končno poročilo

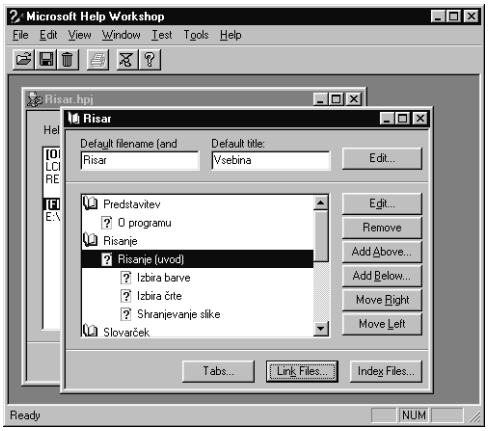


Za učinkovitejšo pomoč lahko pripravimo tudi kazalo. Shranjeno je v posebni datoteki, ki jo prav tako lahko izdelamo z orodjem Help Workshop. Izberemo ukaz **File ➪ New** ter na seznamu potrdimo možnost »Help Contents«.

V novem oknu najprej določimo ime datoteke (polje **Default filename**) V našem zgledu vnesemo »Risar«. Zatem vnesemo privzeti naslov (v polje **Default title**), za nas je to »Vsebina«. Kazalo ustvarjamo tako, da z enim od gumbov **Add** odpremo novo pogovorno okno, v njem izberemo vrsto elementa (vklopimo stikalo **Heading** za naslov ali **Topic** za temo) ter določimo njegove parametre (pri naslovu vnesemo

zgolj vsebino v polje **Title**, pri temi pa moramo poleg naslova v polje **Topic ID** vnesti še njeno enolično oznako. Vnos potrdimo z gumbom OK. Vse dodane elemente hierarhično urejamo z gumbi **Move**.

Slika 10-43: Vsebina datoteke s pomočjo med nastajanjem in med poznejšo rabo



Izdelano kazalo moramo shraniti v datoteko .CNT. Sedaj lahko v osnovnem projektu z gumbom **Options** pod jezičkom **Files** v vnosnem polju **Contents file** določimo zunanjo datoteko vsebine.

> **NAMIG** Če želimo, da se nam ob pregledu pomoči prikaže pripravljeno kazalo vsebine, moramo njeno datoteko (CNT) shraniti v isti mapi kakor prevedeno datoteko s pomočjo (HLP).

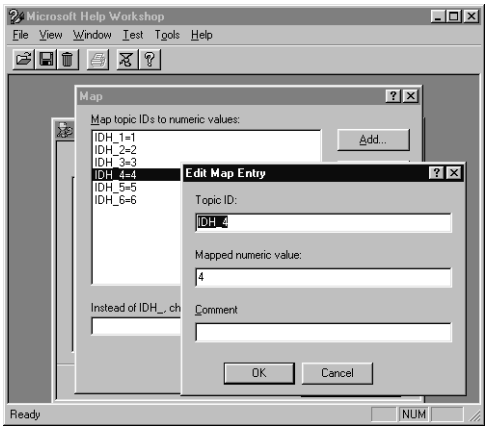
Da lahko pripravljene teme povezujemo z elementi uprabniškega vmesnika v visual basicu, moramo vsako opremiti še z enolično številsko oznako. Povezave med navadno in odgovarjajočo številsko oznako določimo z gumbom **Map**... Prikaže se pogovorno okno, v katerem z gumbom Add dodamo novo oznako, tako da v polje **Topic ID** vnesemo želeno oznako, v polje **Mapped numeric value** pa pripadajočo številko.

Ko določimo ustrezne povezave, datoteko s pomočjo spet prevedemo.

> **NAMIG** Tudi datoteko s projektom in vsebino pomoči najdemo na priloženi plošči CD-ROM v mapi \Primeri\2-Delavnica\3-Risar\4-Pomoč\Help\.

Uporaba običajne pomoči

Za najosnovnejšo rabo pripravljene pomoči v projektu visual basica moramo nastaviti le eno lastnost. Določimo jo lahko v razvojnem okolju ali iz programske

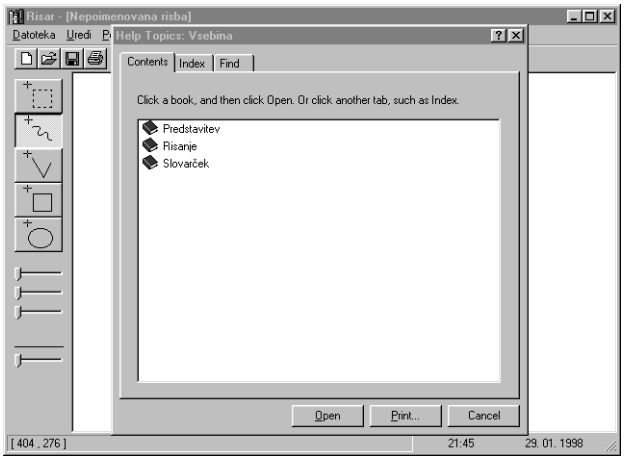


Slika 10-44: Povezovanje tem z enolično številčno oznako. Desno vidimo poznejši učinek.

kode. Nastavitev v razvojnem okolju je manj prilagodljiva, saj se lahko vnesena pot pozneje ne ujema več, zato bomo raje posegli po nastavitvi iz kode. V vzpostavitveno proceduro, odziv na dogodek Load glavnega obrazca, dodamo naslednjo vrstico:

```
App.HelpFile = App.Path & "\Risar.Hlp"
```

In to je vse. Tako določena lastnost bo ob pritisku na tipko F1 prikazala pripravljeno datoteko s pomočjo oziroma njeno kazalo, če je v isti mapi tudi datoteka .CNT.



Slika 10-45: Za samodejni prikaz pomoči ob pritisku na tipko F1 zadošča nastavitev ene same lastnosti.

NAMIG Tipka F1 ne sme biti prirejena kakemu menujskemu ukazu, drugače opisana tehnika ne deluje.

Vsebinska občutljivost

Če želimo, lahko vsak gradnik uporabniškega vmesnika, ki lahko ima vnosno točko, povežemo z ustrezno temo v datoteki s pomočjo. To dosežemo z nastavitvijo lastnosti **HelpContextID**, ki sprejme število. To število mora biti večje od nič in mora biti ena od povezanih številčnih oznak v projektu pomoči (glej zgoraj). Mi smo pripravili naslednje številske oznake:

Številska oznaka	Vsebina pojasnjuje
1	Vrstico z gumbi.
2	Risalna orodja.
3	Izbiro barve.
4	Izbiro debeline.
5	Vrstico stanja.
6	Opis risalne površine.

Če npr. gradniku picRisba nastavimo lastnost HelpContextID na vrednost 6 in med risanjem pritisnemo na tipko F1, prejmemo naslednje pojasnilo:

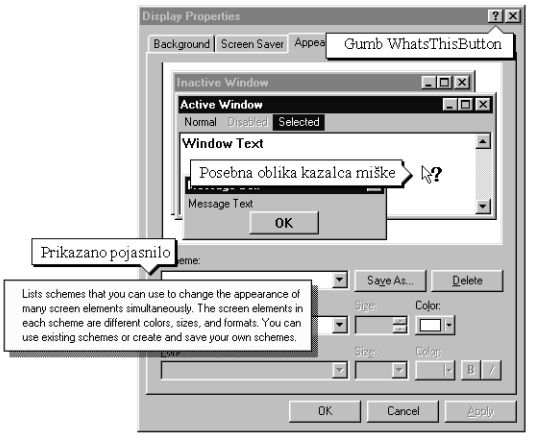
Slika 10-46: Na posamezno temo pomoči se lahko povežemo z nastavitvijo lastnosti HelpContextID.

Kaj pa je to?

Okna 95 so uvedla še en način sklicevanja na teme iz datoteke s pomočjo. Vsako okno lahko preide v posebno stanje pojasnjevanja (poimenovano »What's This Help«). V tem primeru se kazalec miške spremeni v posebno obliko, ki ponazarja zmožnost prikaza pojasnil pomena prikazanih elementov vmesnika. Navadno je oblika tega kazalca puščica z vprašajem.

Če pri dejavnem pojasnjevalnem stanju z miško pokažemo na katerikoli gradnik in ima ta nastavljeno lastnost **WhatsThisHelpID**, se bo prikazalo ustrezno pojasnilo. To pojasnilo ni prikazano znotraj okna pregledovalnika pomoči, ampak se prikaže v manjšem oknu – podobno kakor definicija –, navidezno kot del našega programa.

Slika 10-47: Stanje pojasnjevanja



V posebno stanje je mogoče preiti na različne načine:

- Če obrazcu z obrobo pogovornega okna (lastnost BoderStyle je nastavljena na 3 – Fixed Dialog), nastavimo lastnost **WhatsThisButton** na resnično vrednost (*true*), bo v naslovni vrstici pridobil poseben gumb. Uporabnik ga med delom lahko klikne in s tem preide v stanje pojasnjevanja.
- Obrazec pozna metodo **WhatsThisMode**. Če jo pokličemo, programsko preidemo v stanje pojasnjevanja.
- Če obrazcu nastavimo lastnost **WhatsThisHelp** na resnično vrednost, bo pritisk na tipko F1 sprožil pojasnjevalni in ne običajni način prikaza pomoči.

Na vmesniku Risarja je že pripravljen gumb in ustrezen menujski ukaz, ki nas postavi v pojasnjevalno stanje. Nanju se bomo odzvali s klicem metode WhatsThisMode:

```
Private Sub mnuPomocPojasnila_Click()
  WhatsThisMode
End Sub
```

Da bi pripravljeno pomoč uspešno uporabili, moramo še nastaviti lastnosti WhatsThisHelpID večini gradnikov uporabniškega vmesnika. To lahko storimo med načrtovanjem ali s programsko kodo. Če smo dosledni dosedanjim nastavitvam, bomo stvari znova uredili programsko. Najprej posodobimo vzpostavitev v odzivu na dogodek Load za vse statične gradnike:

```
Private Sub Form_Load()

( ... )

  Toolbar1.WhatsThisHelpID = 1
  StatusBar1.WhatsThisHelpID = 5
  picRisba.WhatsThisHelpID = 6

( ... )

End Sub
```

Dinamično ustvarjeni gradniki tu še niso dostopni, zato zanje poskrbimo ob vzpostavitvi stranske orodjarne:

```
Sub VzpostaviOrodjarno()

( ... )

  optLik(0).WhatsThisHelpID = 3

( ... )

  With sldNastavitve(0)
    .WhatsThisHelpID = 4

( ... )

  With sldNastavitve(3)
    .WhatsThisHelpID = 5

( ... )

End Sub
```

Zdaj smo vzpostavili sodobnejši sistem pomoči. Tipka F1 ne prikliče več kazala sprotne pomoči, temveč aktualno pojasnilo, tako kakor ustrezen gumb v orodjarni ali menujski ukaz **Pomoč ➪ Pojasnila**.

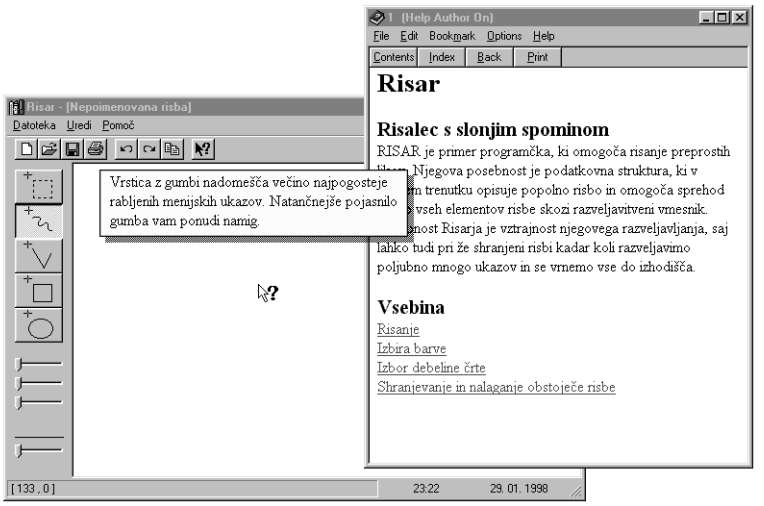
Prikaz kazala sprotne pomoči moramo sedaj rešiti programsko, v odzivu na ustrezen menujski ukaz. Zapisati moramo naslednjo proceduro:

```
Private Sub mnuPomocKazalo_Click()
  dlgDialog.HelpFile = App.HelpFile
  dlgDialog.HelpCommand = cdlHelpContents
  dlgDialog.ShowHelp
End Sub
```

Sistemski prikazovalnik datotek s sprotno pomočjo je mogoče prikazati tudi kot eno izmed standardnih pogovornih oken. Z nastavitvijo nekaj lastnosti natančno določimo, kateri del pomoči želimo priklicati, prikaz okna pa povzroči klic metode **ShowHelp**.

Če bi namesto vsebine želeli prikazati določeno temo pomoči, bi lahko uporabili naslednjo kodo:

```
dlgDialog.HelpFile = App.HelpFile
dlgDialog.HelpContext = "6"
dlgDialog.HelpCommand = cdlHelpContext
dlgDialog.ShowHelp
```



Slika 10-48: Risarju smo dodali zmogljiv sistem sprotne pomoči.

Projekt je s tem skoraj končan. Za konec ga bomo še predelali v prenosljivo obliko.

> **NAMIG** Vso programsko kodo in datoteko s pomočjo ter vse pomožne datoteke najdete na priloženi plošči CD-ROM v mapi \Primeri\ 2-Delavnica\3-Risar\4-Pomoc\.

Razpečevanje

Na žalost uporabnikov in na veselje programskih hiš vsi sodobni programi zahtevajo poseben postopek nameščanja. Utemeljevati, ali je ta postopek sploh potreben, ali gre morda le za zaroto izdelovalcev programske opreme, naj ostane naloga kake druge knjige. Dejstvo je, da je vsak zmogljivejši program, namenjen Oknom, treba pravilno namestiti.

Pri projektih, razvitih v visual basicu, je postopek namestitve *nujen* tudi pri najpreprostejših programčkih, ki jih želimo prenesti v druge računalnike. Če v ciljnem računalniku, v katerem želimo uporabiti razviti program, ni slučajno nameščena popolna različica najzmogljivejšega visual basica, bo našemu prevedenemu programu gotovo manjkala katera od datotek, ki jih nujno potrebuje za svoje izvajanje. Zelo pomembno pravilo, ki ga moramo upoštevati, je zato:

Če program, razvit v visual basicu in preveden v izvršljivo datoteko EXE, deluje v vašem računalniku brez težav tudi zunaj razvojnega okolja, to še ne pomeni, da bo deloval tudi v kakem drugem računalniku.

Če želimo program prenesti v druge računalnike, moramo pripraviti namestitveni postopek (*setup procedure*). Večinoma nam lahko celotno namestitev izdela pripomoček Application Setup Wizard. Če želimo na potek namestitve dodatno vplivati, pa lahko posežemo po orodjih Setup Toolkit. Ker visual basica v slovenščini zaenkrat še ni, je prilagajanje projekta iz orodja Setup Toolkit nujno za pripravo namestitvenih postopkov v domačem jeziku.

NAMIG Ko kupimo visual basic, dobimo dovoljenje za brezplačno razpečevanje vseh projektov, ki jih razvijemo v njem, skupaj s pripadajočimi zunanjimi gradniki. Brez težav lahko razširjamo tudi predložene zglede (vsebino mape Samples\) in bitne slike (mapa Icons\), če smo lastniki boljših različic visual basica (Professional in Enterprise), pa tudi vsebino map Graphics\ in ODBC\. Če pri projektu uporabljamo dodatne zunanje gradnike, ki jih nismo dobili s samim visual basicom, moramo preveriti dovoljenja za razširjanje v njihovi dokumentaciji.

Čarovnik

Če nas uporaba tujega jezika ne moti preveč, nam namestitveni postopek zelo hitro in učinkovito izdela orodje *Application Setup Wizard*. Njegove zmožnosti vključujejo:

- S posebno predlogo visual basica izdela glavni namestitveni program (da, tudi glavni namestitveni program je razvit kar v visual basicu). Ta program poskrbi za uporabniški vmesnik namestitvenega postopka, prepis datotek v ustrezne mape v ciljnem računalniku in zahtevane vnose v sistemski register (*registry*).
- Po potrebi izdela najbolj posodobljeno prevedeno obliko programa, ki ga želimo razpečevati.
- Po želji izdela opis *odvisnosti* projekta.
- Stisne datoteke v obliko, primerno za razpečevanje.
- Po želji razporedi datoteke na zahtevano število disket izbrane velikosti. To lahko stori neposredno na diskete ali v ločene mape na disku.
- Podpira tudi razpečevanje po internetu.

Postopek namestitve precej zapletejo morebitni zunanji gradniki ActiveX, uporabljeni v našem projektu. Ti morajo biti prepisani v posebej določene mape v ciljnem računalniku in zatem prijavljeni v sistemskem registru. Raba pripomočka Application Setup Wizard nam te »umazane« podrobnosti prihrani, moramo po razumeti nekaj izrazov, ki jih ta pripomoček uporablja pri pojasnjevanju in vprašanjih.

Odvisnosti

Opis odvisnosti (*dependency*) je datoteka s seznamom *izvajalnih zahtev* gradnika ali programa. Izvajalne zahteve (*runtime requirements*) vključujejo: seznam potrebnih datotek, ciljno mapo, v katero morajo biti nameščene, postopek njihove napovedi v sistemskem registru in še nekatere dodatne informacije. Če npr. v našem projektu uporabljamo standardna pogovorna okna, je njegovo pravilno izvajanje odvisno od ustrezno nameščene datoteke Comdlg32.OCX.

Pripomoček Application Setup Wizard lahko pripravi opis odvisnosti našega projekta, ki temelji na treh virih:

- Izvajalne zahteve visual basica. Te opisujejo odvisnosti vseh gradnikov pri nameščanju visual basica. Shranjene so v datoteki Vb5dep.INI, ki jo najdete v mapi SetupKit\Kitfil32\.
- Izvajalne zahteve dodatnih gradnikov, uporabljenih v projektu. Vsak posamezen gradnik ima opisane lastne odvisnosti v posebni, njemu pripadajoči datoteki .DEP. Za vse gradnike iz »škatle« visual basica so tako odvisnosti opisane na dveh krajih, drugače pa je z gradniki, ki jih naknadno dodajamo v razvojno okolje.
- Izvajalne zahteve samega projekta. Te ustvari orodje Application Setup Wizard s sestavljanjem odvisnosti visual basica in uporabljenih gradnikov.

Pri projektih, ki smo jih razvili doslej, bi zahteve samega projekta ne vključevale ničesar novega, česar ni mogoče najti v prvih dveh virih. Drugače bo takrat, ko bomo razvijali lastne gradnike. Temu je posvečen četrti del knjige, v katerem si bomo izvajalne zahteve znova podrobneje ogledali.

Orodje Application Setup Wizard poizkuša pripraviti opis odvisnosti z združevanjem informacij iz vseh treh virov. Če za posamezno izvajalno zahtevo v enem izmed naštetih virov ni mogoče najti podrobnejšega opisa, nas na to opozori. Kljub manjkajoči zahtevi lahko dokonča svoje delo, vendar potem morda nameščeni program ne bo pravilno deloval. Opozorilo lahko zanemarimo samo, če smo prepričani, da je v ciljnem računalniku ustrezen zunanji gradnik že pravilno nameščen in preizkušeno deluje.

Čaranje

Tudi pripomoček Application Setup Wizard je zelo podoben drugim čarovnikom, ki so nam na voljo. Skozi zaporedje pogovornih oken nas pripelje do konca sicer zapletenega opravila. Ne moremo ga uporabiti neposredno v razvojnem okolju, ampak ga poženemo ločeno. Zato, preden ga uporabimo, še zadnjič shranimo dokončani projekt in končamo delo v visual basicu. Ikona čarovnika Application Setup je v isti skupini kakor drugi elementi visual basica. Ob zagonu nas pozdravi

uvodni zaslon z nekaj pojasnili. Po želji lahko njegovo prikazovanje v prihodnje ukinemo z vklopom stikala **Skip this screen in the Future**.

Slika 10-49: Uvodni pozdrav

Ko kliknemo gumb **Next**, se premaknemo k prvemu koraku, v katerem izberemo želeni projekt. V vnosno polje vnesemo ime datoteke z opisom izbranega projekta (datoteka ima podaljšek .VBP). Najlaže jo najdemo z gumbom **Browse**, ki nam omogoči brskanje po datotečnem sistemu. Z vklopom stikala **Rebuild the Project** določimo, naj čarovnik projekt prevede v izvršljivo datoteko. Če tega pri posodobljeni različici projekta še nismo storili, je stikalo nujno treba vključiti. Mi uporabimo projekt Risar.VBP in vključimo prevajanje.

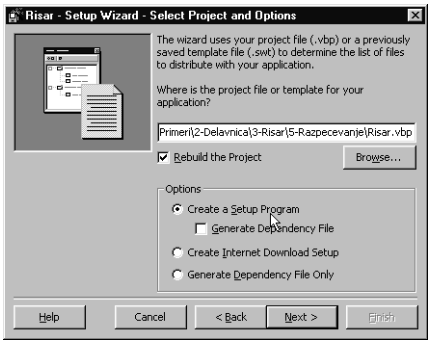
V skupini **Options** lahko izberemo tri različna opravila:

- Pripravo projekta za razpečevanje prek zunanjega medija (diskete, CD-ROM itd.). Vklopimo stikalo **Create a Setup Program**. V tem primeru lahko vklopimo tudi stikalo **Generate Dependency File**, če želimo pripraviti še opis odvisnosti samega projekta[42]. Opis odvisnosti, ki je potreben za pravilno *namestitev* projekta, se ustvari v vsakem primeru.
- Pripravo projekta za razpečevanje prek interneta. Ta možnost je na voljo za projekte, ki določajo nove gradnike ActiveX. Vklopimo stikalo **Create Internet Download Setup**. Ob izbiri tega stikala se levo v oknu prikaže gumb, ki nas poveže s spletno stranjo na naslovu `http://www.microsoft.com/vbasic/icompdown/`. Na njej so najnovejše informacije o Microsoftovem načinu razpečevanja izdelanih projektov prek omrežja.
- Izdelavo datoteke odvisnosti s seznamom izvajalnih zahtev projekta. Vklopimo stikalo **Generate Dependency File Only**. Ta možnost je najbolj uporabna pri projektih, ki določajo nove, v visual basicu razvite gradnike. Zaenkrat je še ne bomo uporabljali.

[42] Ta odvisnost je pomembna pri projektih, ki so sestavni del večjih celot, npr. projekta z novim gradnikom ActiveX. V splošnem ni nobene škode, če opis odvisnosti ustvarimo za vsak projekt.

Mi namestitve prek interneta ne moremo izbrati. Na voljo je le za lastne gradnike ActiveX, ki jih bomo razvijali v četrtem delu knjige. Za izdelavo običajnega namestitvenega postopka pustimo vključeno prvo stikalo (Create a Setup Program), stikalo Generate Dependecy File pa naj ostane izključeno.

Slika 10-50: Določanje projekta



Če smo izbrali prevajanje projekta, bo prehod na naslednji korak nekoliko daljši, saj bo čarovnik v ozadju pognal visual basic in mu naložil prevajanje izbranega projekta. V drugem koraku izberemo način porazdelitve pripravljenih datotek. Izbiramo lahko med:

- Disketami (stikalo **Floppy Disk**). V tem primeru se bodo pripravljene datoteke shranile neposredno na ustrezno število disket.
- Mapo na disku (stikalo **Single Directory**). Za razpečevanje s ploščo CD-ROM ali zmogljivim zunanjim nosilcem (npr. diskete Zip) je najprimernejše shranjevanje pripravljenih datotek v eno samo mapo na disku.
- Več map na disku (stikalo **Disk Directories**). To je uporabno predvsem za poznejšo pripravo disket. Če bi izbrali prvo stikalo, moramo imeti pripravljeno ustrezno število praznih disket, prepisovanje pa bo dolgotrajno in manj zanesljivo. Bolje je pripravljene datoteke porazdeliti v več map na disku in diskete pripraviti pozneje (npr. s pripomočkom Okenski komandant).

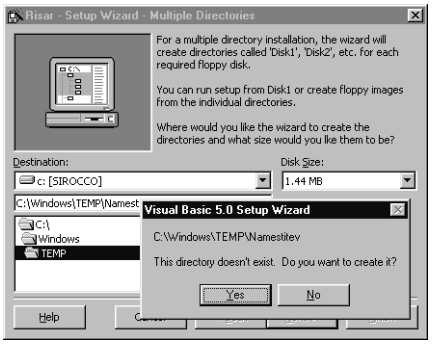
> **NAMIG** Namestitveni program, ki ga bo pripravil Application Setup Wizard, je izdelan tako, da se znajde pri uporabi disket ali ustrezno poimenovanih map na disku. Če pripravimo namestitveni postopek v več mapah na disku, ga lahko brez težav preizkusimo z zagonom namestitvenega programa v prvi mapi (poimenovana je Disk1\).

Mi izberimo zadnje stikalo (Disk Directories) in se premaknimo k naslednjemu koraku.

Slika 10-51: Izbira načina porazdelitve pripravljenih datotek

V četrtem koraku moramo določiti ciljno mapo, v katero se bodo shranile pripravljene datoteke, in morebitno velikost zunanjega medija. Velikost izberemo z zaprtega seznama **Disk Size**, na katerem so zbrane vse standardne zmogljivosti 3,5-palčnih disket. Pot lahko izberemo z uporabo seznamov, lahko pa vnesemo tudi ime še neobstoječe mape kar neposredno v vnosno polje. Potem nas ob prehodu na nov korak čarovnik opozori in po želji samodejno ustvari novo mapo. Če smo določili način porazdelitve pripravljenih datotek v več map, bo čarovnik v ciljni mapi samodejno ustvaril podmape Disk1, Disk2 itn. in vanje razvrstil pripravljene datoteke.

Slika 10-52: Določitev ciljne mape

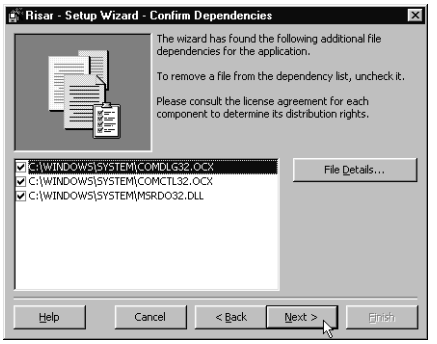


V naslednjem koraku lahko določimo strežnike ActiveX, ki jih naš projekt morda uporablja. Zaenkrat s strežniki ActiveX še nismo imeli opravka, zato ta korak preskočimo.

Korak, ki sledi, ponudi seznam zunanjih datotek, od katerih je odvisno pravilno izvajanje našega projekta (zunanje odvisnosti). Začetni seznam oblikuje sam čarovnik na podlagi analize projektne datoteke. Če smo prepričani, da je določena datoteka v ciljem računalniku že nameščena, lahko njej ustrezno stikalo izklopimo. Mi pustimo stvari nedotaknjene in se preselimo k predzadnjemu koraku.

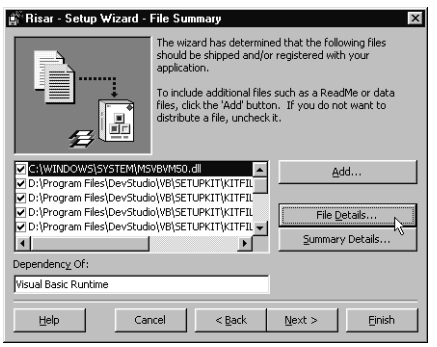
Slika 10-53: Tega koraka zaenkrat ne potrebujemo.

Slika 10-54: Zunanje odvisnosti



Na tem mestu čarovnik analizira odvisnosti vseh vključenih enot. Po končani analizi nam predstavi seznam datotek, ki so potrebne za uspešno namestitev projekta v druge računalnike. Za vsako datoteko, ki jo izberemo s seznama – razen same prevedene datoteke projekta –, je na kombiniranem seznamu **Dependency Of** mogoče najti elemente, ki so od nje odvisni in jo zahtevajo. Z gumbom **File Details**... lahko prikažemo natančnejši opis izbrane datoteke in se potem odločimo, ali to datoteko resnično potrebujemo. Na splošno je čarovnik dovolj natančen in izklop posameznih datotek ni priporočljiv, razen če natančno vemo, kaj počnemo. Skupno statistiko namestitvenih datotek nam prikaže gumb **Summary Details**...

Slika 10-55: Seznam datotek, potrebnih za uspešno namestitev

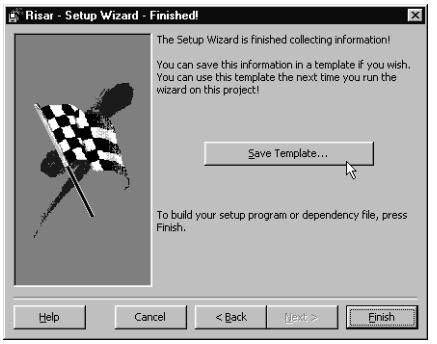


Seveda pa čarovnik ni vsemogočen. Če naš projekt potrebuje še sprotno pomoč v zunanjih datotekah, zunanje podatkovne datoteke ali če bi radi v namestitveni postopek vključili še neodvisne datoteke za posebne namene (npr. datoteka z opisom zadnjih sprememb v projektu), moramo želene datoteke na seznam dodati ročno. Tu nam pomaga gumb **Add**... Prikaže standardno pogovorno okno za odpiranje datotek, s katerim lahko pobrskamo po datotečnem sistemu in najdemo vse dodatne datoteke. Mi bomo morali dodati zunanji datoteki s pomočjo, Risar.HLP in Risar.CNT. Vse naknadno dodane datoteke se po privzeti nastavitvi namestijo v isto mapo kakor prevedena datoteka projekta. To mapo bo izbral uporabnik namestitvenega postopka.

> **NAMIG** Če imamo posebne želje, lahko izberemo drugačen cilj datotek na seznamu. S pogovornim oknom gumba **File Details**..., ki prikaže podrobnosti posamezne datoteke, lahko na seznamu **Destination Directory** določimo ciljno mapo. Mogoče je vnesti absolutno pot (npr. »c:\«) ali pa izberemo simbolično ime ene od sistemskih poti (opisane so v obliki `$(<simbolično ime>)`). To nastavitev zelo redko določamo sami.

S tem smo določanje parametrov, potrebnih za pripravo namestitvenega postopka, končali in se lahko premaknemo k zadnjemu koraku. V zadnjem koraku nam čarovnik omogoča, da nastavljene parametre trajno shranimo kot predlogo, kar nam bo olajšalo izdelovanje namestitvenih postopkov z morebiti izboljšano različico programa. Ko kliknemo gumb **Save Template**..., lahko izberemo datoteko, ki bo te nastavitve hranila.

Slika 10-56: Končni korak čarovnika



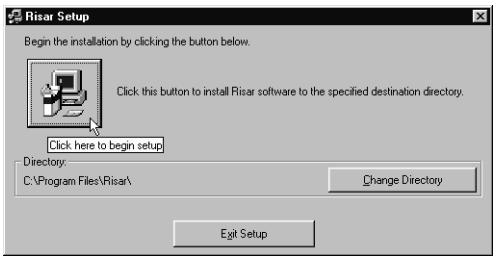
Ko kliknemo gumb **Finish,** bo čarovnik začel stiskati in razporejati izbrane datoteke v ciljne mape. Pri tem bo upošteval zmogljivost ciljnega medija (če smo jo določili). Poleg datotek, ki so bile naštete na seznamu predzadnjega koraka, bo končna različica vsebovala še nekaj dodatnih, potrebnih za vzpostavitev postopka namestitve. Ena od njih, Setup.EXE, je začetek vsake namestitve. Najdemo jo

v podmapi Disk1\ znotraj izbrane ciljne mape. Z njo pravkar pripravljeni postopek namestitve tudi preizkusimo.

> **NAMIG** Datoteko Setup.EXE lahko po želji preimenujemo, npr. v Namesti.EXE.

Namestitveni program ponuja uporabniku izbor ciljne mape, poskrbi za vse potrebne postopke namestitve in omogoča poznejšo odstranitev projekta s standardnim sistemskim pripomočkom.

Slika 10-57: Izdelana namestitev v rabi



Če smo shranili predlogo namestitve in projekt pozneje posodobimo, lahko novo namestitev izdelamo zelo hitro. V prvem koraku čarovnika Application Setup Wizard ne izberemo več projektne datoteke visual basica, temveč kar shranjeno predlogo namestitve. Tovrstne predloge se shranijo v datoteko s podaljškom SWT. Če v prvem koraku izberemo datoteko s predlogo, lahko nemudoma kliknemo gumb **Finish** in namestitveni postopek se bo pripravil po shranjenih nastavitvah, ali pa se sprehodimo po korakih in vnesemo morebitne popravke.

Orodja

Če nam katerakoli zmožnost pripomočka Application Setup Wizard ne ustreza povsem, lahko uporabimo orodja Setup Toolkit, ki so v mapi SetupKit\. Z njimi lahko ročno izdelamo ves postopek namestitve. Seveda je to skrajnost, h kateri se le redko zatečemo. Pogosto zadošča, da prenovimo le delček privzete nastavitve in znova uporabimo čarovnik Application Setup.

Potek namestitve

Preden si ogledamo posamezna orodja, moramo natančneje spoznati postopek namestitve. V njem sodelujejo naslednje datoteke:

Datoteka	Opis
Setup.EXE	Vstopna točka v namestitveni postopek, ki poskrbi za njegovo vzpostavitev.
Setup1.EXE	Uporabniški program namestitve, razvit v visual basicu, ki ga lahko posodobimo.
Setup.LST	Datoteka z besedilom, ki natančno opisuje postopek namestitve. Uporabljata jo oba zgoraj omenjena programa.
Vb5stkit.DLL	Zunanja knjižnica posebnih podprogramov, ki jih uporablja uporabniški program namestitve Setup1.EXE.
St5unst.EXE	Pripomoček, ki omogoča odstranitev nameščenega programa iz ciljnega računalnika.

Vsaka namestitev se začne z zagonom programa Setup.EXE, ki ga po želji lahko tudi preimenujemo, vendar na njegov videz ne moremo vplivati. Njegova naloga je vzpostaviti postopek namestitve. Najprej razširi in prepiše osnovne potrebne datoteke namestitve, zatem pa prepusti delo programu Setup1.EXE.

Najvplivnejši element bodoče nastavitve je projekt Setup1, na katerem temelji uporabniški vmesnik, in potek postopka namestitve, določen s programom Setup1.EXE. Projekt je shranjen v mapi SetupKit\Setup1\. Povsem enakovreden je kateremukoli drugemu projektu v visual basicu in ga lahko prikrojimo lastnim potrebam. Preden se lotimo popravljanja tega projekta, si pripravimo rezervno kopijo obstoječega stanja, saj bodo spremembe vplivale na vsak postopek namestitve, ki bo pozneje izdelan z orodjem Application Setup Wizard. Če ga kje polomimo, imamo možnost hitro obnoviti prejšnje stanje.

> **NAMIG** Uporabniki Visual Basica Professional ali Enterprise imate rezervno kopijo vseh nameščenih datotek tudi na plošči CD-ROM, v mapi VB\.

Datoteka Setup.LST obsega opis vseh spremenljivih parametrov namestitvenega postopka, zato je program Setup1.EXE vsestransko uporaben za katerokoli namestitev. Njeno vsebino določi pripomoček Application Setup Wizard, po želji pa jo lahko tudi ročno posodobimo. To opravilo lahko pripelje do nepredvidljivih posledic, zato je primerno le za najbolj prekaljene programerje. Opis vsebine najpomembnejših datotek – tudi datoteke Setup.LST – najdete v Dodatku B.

Če bomo na potek namestitve ročno vplivali, si lahko pomagamo še z orodji za ročno stiskanje poljubne datoteke. Na voljo sta dva konzolna programa, ki ju najdemo v mapi SetupKit\Kitfil32\. Za stiskanje posamezne datoteke je na voljo pripomoček Compress.EXE. Pri izdelovanju paketov novih gradnikov za razpečevanje prek interneta je na voljo še konzolni program Makecab.EXE, ki zna več datotek shraniti v stisnjeno datoteko CAB (*Cabinet File*).

Vaje

Ker smo že pravi strokovnjaki za visual basic, je tu nekaj vaj za poznavalce našega kova, ki lahko precej izboljšajo zmožnosti Risarja:

1. Popestrite izbiro barv še z možnostjo izbire iz standardnega pogovornega okna.
2. Dodajte upoštevanje označenega dela slike pri bitnem izvozu slik, podobno kakor se upošteva pri prenosu na odložišče. Prenos na odložišče opremite še s prestrezovalnikom izvajalne napake (pogosta napaka, ki se lahko pripeti, je pomanjkanje pomnilnika).
3. Omogočite izpis risbe na poljuben tiskalnik.
4. Med funkcijami Windows API najdete funkcijo FloodFill, ki zna zapolniti omejena območja s poljubno barvo. Dodajte novo risalno orodje, ki omogoča zapolnjevanje s to funkcijo.
5. Povečajte risalno površino z običajnimi drsniki ob robu okna. Uporabnik naj ima možnost določiti začetno velikost slike. Drsnika naj spreminjata njeno izhodišče, procedura za obnavljanje slike pa naj to izhodišče smiselno uporabi. Spretnejši pripravite še ukaze za povečanje in pomanjšanje slike.
6. Prevedite namestitveni postopek v slovenščino!

TRETJI DEL

Podatki

V TEM DELU

Uvod v zbirke
Predmetni model
Jezik SQL
Strežnik in odjemalec
Razpečevanje

Računalniki so naprave za samodejno obdelovanje podatkov. Če je teh podatkov več in jih je mogoče urediti v strukturirano obliko, je smiselno uporabiti *zbirko podatkov*. Zbirke podatkov tako najdemo v preprostih programih, kot je domača knjiga receptov, ali v bolj zapletenih programih, kot je večpredstavna enciklopedija. Brez pretiravanja bi lahko rekli, da zmogljivejšega programa brez uporabe take ali drugačne zbirke podatkov ne moremo razviti.

Uporaba zbirk podatkov (*databases*) si zasluži poseben del knjige. Gre za eno najbolj zmogljivih in najbolj uporabnih zmožnosti visual basica, s katero lahko pripravimo tudi zmogljive poslovne programe. Vse od vključitve podatkovnega stroja JET (*Joint Engine Technology*, saj veste, da so programerji v glavnem hipiji), ki si jo visual basic od različice 3.0 naprej deli z uporabniško zbirko podatkov Microsoft Access, je njegovim uporabnikom na voljo moč, ki je že skoraj prevelika za učinkovito obvladovanje. Pogosto je pri enem opravilu mogoče ubrati več poti, ki pa niso enako učinkovite. Napotki v nadaljevanju bodo zato marsikomu prišli prav.

11. POGLAVJE

Uvod v zbirke

Raba zbirk podatkov je v visual basicu lahko zelo preprosta. Za osnovno pregledovanje podatkov lahko uporabimo pripravljene, vgrajene gradnike, ki jim nastavimo nekaj lastnosti. Delujoč program dobimo, ne da bi zapisali eno samo vrstico kode. Z nadaljnjo uporabo gradnikov, pisanjem odzivnih procedur za nekatere dogodke, še posebej pa z rabo posebnega *predmetnega modela* lahko zbirke podatkov popolnoma obvladujemo. Za začetek si bomo ogledali splošen opis dostopa do podatkov in rabo vgrajenih gradnikov. Predmetni model bomo natančneje spoznali v naslednjem poglavju.

Dostop do podatkov

Visual basic omogoča dva različna načina povezovanja z zbirkami podatkov:

- Raba posebnih gradnikov (*data acces* in *data bound controls*).
- Raba predmetov DAO (*Data Access Objects*).

Medtem ko prvi način omogoča pregledovanje in popravljanje podatkov brez ene same vrstice kode, drugi način omogoča popoln nadzor zbirke podatkov samo iz programske kode. Oba načina se med seboj ne izključujeta, prav nasprotno. Pogosto kombiniramo prednosti obeh, da dobimo kar najučinkovitejši program.

Ne glede na izbrani način povezovanja smo vedno izolirani od fizičnega dela z datotekami. Tega večinoma za nas postori *podatkovni stroj* »Jet«. Podatkovni stroj (*database engine*) je knjižnica podprogramov, ki fizični zapis podatkov preslikajo v programske predmete, ki jih nadziramo iz kode. Prednost stroja Jet je v enolični predstavitvi najrazličnejših virov podatkov. Za programerja je skoraj vseeno, ali dela s podatki, zapisanimi v običajni datoteki z besedilom, ali uporablja zmogljiv podatkovni strežnik. *Predmetni model*, ki ga pri tem uporablja, je enak. Podatkovni stroj omogoča obdelovanje podatkov iz naslednjih virov:

- Domorodne zbirke (*native databases*). Zbirka podatkov, zapisana v obliki Microsoft Access (.MDB), je za uporabo v visual basicu prek podatkovnega stroja Jet najbolj učinkovita.

- Zunanje zbirke (*external databases*). Podatki, zapisani v zaporednih, indeksiranih datotekah[43] v enem izmed priljubljenih oblik zapisa, kot so dBASE, Btrieve ali Paradox, so prav tako dostopni, vendar je njihova raba manj učinkovita. Med zunanje zbirke prištevamo tudi datoteke z besedilom in nekatere preglednice (v oblikah zapisa Microsoft Excel in Lotus 1-2-3).
- Zbirke ODBC (*Open Database Connectivity*). Zunanji podatkovni strežniki, ki spoštujejo protokol ODBC, so prav tako dostopni. Protokol ODBC se je uveljavil kot standard, zato ga podpira velika večina sodobnih podatkovnih strežnikov, kot so Microsoft SQL Server, Oracle, Informix itn.

> **NAMIG** Na podatkovne strežnike, ki spoštujejo protokol ODBC, se je mogoče priklopiti tudi neposredno, brez vmešavanja podatkovnega stroja Jet. V tem primeru pridobimo hitrost, izgubimo pa prilagodljivost in prenosljivost kode, saj moramo upoštevati značilnosti neposrednega dela s podatkovnim virom.

Zmogljivosti podatkovnega stroja so odvisne od različice visual basica, ki jo uporabljamo. Najpreprostejša, Learning Edition, vključuje le omejen nabor funkcij, ki ne omogoča ustvarjanja novih zbirk podatkov ali predmetov za njihovo obravnavo. Za rabo zbirk podatkov sta tako praktično uporabni le boljši različici, Professional in Enterprise. Najzmogljivejša različica je posebej namenjena delu z oddaljenimi podatkovnimi viri po načelu odjemalca in strežnika, vse druge primere pa enako dobro obvladuje tudi različica Professional.

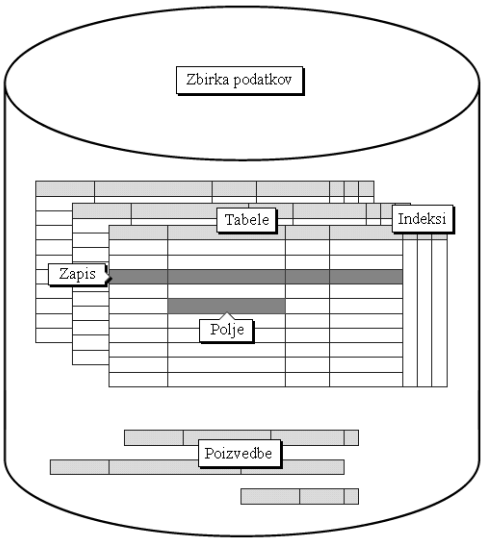
Zgradba zbirk podatkov

Oglejmo si nekaj izrazov, ki nas bodo spremljali pri tem delu. Njihovo poznavanje je bistvenega pomena za razumevanje nadaljevanja.

Zbirka podatkov (*database*) je ponavadi ena ali več datotek z večjo količino smiselno urejenih podatkov, pa tudi orodij za njihovo obdelovanje. Podatki so znotraj zbirke urejeni v *tabelah*. Tabela (*table*) je skupina istovrstnih *zapisov*, zapis (*record*) pa je sestavljen iz enega ali več *polj*. Polje (*field*) je podatek osnovnega podatkovnega tipa. Zapisom tabele pogosto rečemo tudi *vrstice*, poljem pa *stolpci* tabele. V določenem trenutku je neposredno dostopna samo ena vrstica tabele. Nanjo kaže podatkovni *kazalec* (*cursor*). Zapis v tej dejavni vrstici imenujemo tudi *trenutni* zapis (*current record*). Če želimo doseči kak drug zapis, moramo najprej prestaviti podatkovni kazalec.

[43] Tovrstne datoteke imenujemo tudi datoteke ISAM (*Indexed Sequential Access Method*).

Slika 11-1: Zgradba zbirke podatkov



Ker so zbirke podatkov samo posebna vrsta datotek, so zapisi v tabelah urejeni v *naravnem* vrstnem redu, tako kakor so bili dodani. Če bi bilo treba po vsakem dodajanju vso datoteko urediti, bi to vzelo silno veliko časa, hkrati pa bi imeli le en način urejanja podatkov. Urejenost podatkov v tabeli zato določajo posebni seznami, ki vključujejo le zaporedje zapisov, urejenih po določenem merilu. Tak seznam imenujemo *indeks*. Tabela, ki ima enega ali več indeksov, omogoča nadvse hitro preiskovanje, tabele brez indeksov pa je mogoče preiskovati le zaporedno (*sequential*), zapis za zapisom, od začetka do konca.

Zbirke podatkov lahko – to je predvsem odvisno od oblike, kako so zapisane – vključujejo še orodja za obdelovanje tabel. To so *poizvedbe*, *pogledi*, *poročila* in podobno. Poizvedba (*query*) je ukaz, ki iz ene ali več tabel izlušči novo tabelo s podatki, ki nas zanimajo, urejenimi po zahtevanem vrstnem redu. Pogled (*view*) je kombinacija tabel in poizvedb, poročilo (*report*) pa je pogled, urejen za oblikovan izpis.

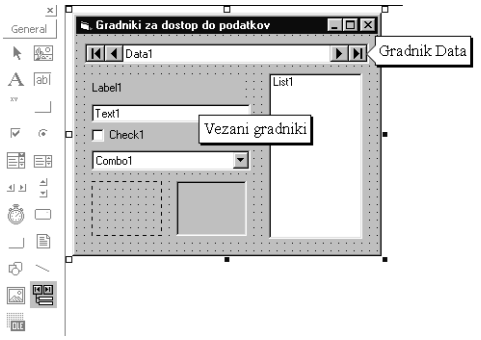
Ker se v kodi visual basica lahko ukvarjamo tako s tabelami kakor z izidi poizvedb, se je uveljavil izraz *zbirka zapisov* (*recordset*), ki lahko predstavlja kakršnokoli skupino zapisov, bodisi iz ene same tabele, bodisi izbor zapisov več tabel iz *relacijske* poizvedbe.

Gradniki za dostop do podatkov

Doslej smo že dvakrat omenili, da je z uporabo gradnikov mogoče narediti program za pregledovanje in popravljanje podatkov brez zapisa ene same vrstice kode. To pa

ni vse, kar nam gradniki omogočajo. Z nekaj odzivnimi procedurami lahko določimo zmogljiv program, ki omogoča pregledovanje, popravljanje, dodajanje in brisanje zapisov.

Slika 11-2: Gradnik Data in vezani gradniki med načrtovanjem

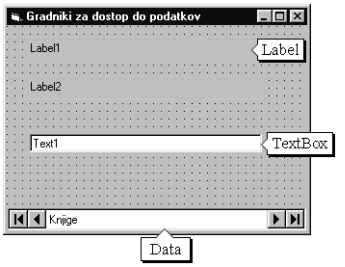


Med vgrajenimi gradniki, ki jih poznajo vse različice visual basica, najdemo tudi gradnik »podatki« (*Data*). Z njim se lahko navežemo na poljuben podatkovni vir, ki ga podpira podatkovni stroj Jet. Gradnik poleg tega omogoča osnovno navigacijo po podatkih, pozna pa tudi nekaj lastnosti, metod in dogodkov, ki omogočajo razvoj zmogljivejšega programa. Z gradnikom Data podatkov še ne moremo pregledovati. Pregledovanje nam omogočijo *vezani gradniki*, katerih raba je mogoča šele po navezavi na podatkovni vir. Vezani gradniki (*bound controls*) so zgolj podmnožica obstoječih gradnikov, kot sta npr. vnosno polje ali stikalo, ki poznajo lastnost DataSource.

Brskanje brez kode

Za prikaz zmožnosti gradnikov si pripravimo preprost zgled. Odprimo nov projekt vrste »Standard EXE« in oblikujmo enostaven uporabniški vmesnik po zgledu naslednje slike:

Slika 11-3: Zgled rabe gradnikov: uporabniški vmesnik



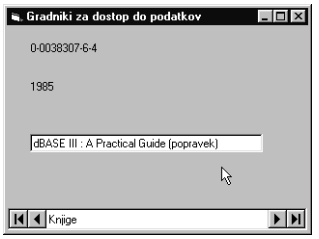
Seznam lastnosti, ki jih moramo nastaviti, je zbran v naslednji tabeli:

Gradnik	Lastnost	Nastavitev
podatki (*Data*)	(Name)	Data1
	DatabaseName	biblio.mdb
	RecordSource	Titles
	Caption	Knjige
	Align	2 - Align Bottom
oznaka (*Label*)	(Name)	Label1
	DataSource	Data1
	DataField	ISBN
oznaka (*Label*)	(Name)	Label2
	DataSource	Data1
	DataField	Year Published
vnosno polje (*TextBox*)	(Name)	Text1
	DataSource	Data1
	DataField	Title

Za zbirko podatkov smo uporabili datoteko Biblio.MDB, ki je priložena visual basicu. V njej najdemo podatke o različnih knjigah.

> **NAMIG** Kodo iz tretjega dela knjige najdete na priloženi plošči CD-ROM, v mapi \Primeri\3-Podatki\.

To je vse. Nobene kode. Žal učinek nastavitev ni opazen že med načrtovanjem, zato moramo za preizkus program pognati. Tudi brez posebne kode se lahko pomikamo po zapisih in trenutni zapis tudi popravimo. Popravljanje nam omogoča vnosno polje. Če v njem popravimo naslov knjige *in* se premaknemo na nov zapis, zatem pa se vrnemo nazaj, bomo opazili, da se je sprememba ohranila. Za dokončen preizkus lahko program zaustavimo in znova poženemo. Ker so se vse spremembe zapisale neposredno v zbirko podatkov, bodo veljale, kadarkoli poženemo program.



Slika 11-4: Zgled v rabi

NAMIG S tem, ko vsebino določenih polj prikazujemo v oznaki, preprečimo njihovo spreminjanje s strani uporabnika. Nismo pa onemogočili spreminjanja vrednosti iz programske kode. Če vezani oznaki programsko spremenimo lastnost Caption in se premaknemo na nov zapis, bomo prav tako trajno popravili vrednost polja prejšnjega zapisa. Preprečevanje shranjevanja sprememb elegantno rešimo z odzivno proceduro za dogodek Validate, ki ga bomo spoznali v nadaljevanju.

Lastnosti

Po kratki demonstraciji preprostosti rabe gradnikov za dostop do podatkov si natančneje oglejmo pomembnejše lastnosti.

Lastnosti **DatabaseName** in **RecordSource**

Navezava na domorodne zbirke podatkov (oblika Microsoft Access) z gradnikom Data je mogoča zgolj z nastavitvijo teh dveh lastnosti. Lastnost DatabaseName mora biti nastavljena na popolno pot do datoteke z zbirko podatkov, lastnost RecordSource pa mora določati veljavno *tabelo* ali *poizvedbo*. Pri poizvedbah lahko izbiramo med *poimenovanimi* poizvedbami, ki so že shranjene v zbirke podatkov, ali pa poizvedbo zapišemo neposredno, v obliki stavka jezika *SQL* (*Structured Query Language*), ki ga bomo spoznali malo pozneje. Gradnik Data tako lahko predstavlja vse zapise iz posamezne tabele ali pa urejen izbor zapisov iz tabel (izid poizvedbe).

Obe lastnosti je mogoče določiti med načrtovanjem ali iz programske kode. Načrtovanje je boljše, ker nam takoj po nastavitvi lastnosti DatabaseName visual basic že ponudi seznam tabel in poizvedb, ki so ena izmed razpoložljivih možnosti za nastavitev lastnosti RecordSource. Po drugi strani je med načrtovanjem včasih težko opredeliti, kje natančno bo zbirka podatkov. Med izvajanjem lahko poizvedujemo po mapi, v kateri je program, z lastnostjo Path vgrajenega predmeta App:

```
Private Sub Form_Load()
  Data1.DatabaseName = App.Path & "\podatki\Zbirka.MDB"
  Data1.RecordSource = "Tabela1"
End Sub
```

Lastnost **Connect**

Z lastnostjo Connect lahko uporabimo podatke v zunanjih zbirkah. Tudi to lastnost lahko nastavimo iz programske kode. Ker njena nastavitev neposredno vpliva na pomen lastnosti DatabaseName, jo moramo nastaviti najprej. Lastnost je niz, ki je za domorodne zbirke lahko prazen, pri drugih pa določa obliko zapisa in morebitne dodatne parametre, ki jih naštejemo na seznamu, ločenem s podpičjem (;).

Medsebojno odvisnost lastnosti Connect in DatabaseName nakazuje naslednja tabela:

Oblika zapisa zbirke	Lastnost Connect	Lastnost DatabaseName	Zgled
domorodna zbirka	prazen niz (privzeto)	pot do datoteke	c:\pot\zbirka.mdb
dBASE III	`dBASE III;`	pot do mape z datotekami	c:\pot
dBASE IV	`dBASE IV;`	pot do mape z datotekami	c:\pot
dBASE 5	`dBASE 5.0;`	pot do mape z datotekami	c:\pot
FoxPro 2.0	`FoxPro 2.0;`	pot do mape z datotekami	c:\pot
FoxPro 2.5	`FoxPro 2.5;`	pot do mape z datotekami	c:\pot
FoxPro 2.6	`FoxPro 2.6;`	pot do mape z datotekami	c:\pot
Paradox 3.x	`Paradox 3.x;`	pot do mape z datotekami	c:\pot
Paradox 4.x	`Paradox 4.x;`	pot do mape z datotekami	c:\pot
Paradox 5.x	`Paradox 5.x;`	pot do mape z datotekami	c:\pot
Excel 3.0	`Excel 3.0;`	pot do datoteke	c:\pot\preglednica.xls
Excel 4.0	`Excel 4.0;`	pot do datoteke	c:\pot\preglednica.xls
Excel 5.0 / 95	`Excel 5.0;`	pot do datoteke	c:\pot\preglednica.xls
Excel 97	`Excel 97;`	pot do datoteke	c:\pot\preglednica.xls
uvoz HTML	`HTML Import;`	pot do datoteke	c:\pot\stran.html
izvoz HTML	`HTML Export;`	pot do mape z datotekami	c:\pot
besedilo	`Text;`	pot do mape z datotekami	c:\pot

vir ODBC	`ODBC;`
	`[DATABASE=`*`zbirka`*`;]`
	`[UID=`*`uporabnik`*`;]`
	`[PWD=`*`geslo`*`;]`
	`[DSN=`*`sistemskiVir`*`;]`
	`[LOGINTIMEOUT=`*`iztekPrijave`*`;]`
		(nič)	–

vir Exchange	`Exchange;`
	`MAPILEVEL=`*`mapa`*`;`
	`[TABLETYPE={ 0 | 1 }];`
	`[PROFILE=`*`profil`*`;]`
	`[PWD=geslo;]`
	`[DATABASE=`*`zbirka`*`;]`
		pot do datoteke	c:\pot\zbirka.mdb

Vidimo, da podatkovni stroj Jet resnično podpira široko paleto najrazličnejših oblik zapisa podatkov. Pri oblikah ODBC in Exchange lahko v lastnosti Connect navedemo še cel kup dodatnih parametrov. Večina je, kakor vidimo, neobveznih. Če jih izpustimo, a so potrebni za vzpostavitev povezave (npr. geslo), se samodejno pokažejo dodatna pogovorna okna, ki zahtevajo vnos manjkajočega parametra.

NAMIG Pot do podatkovnega vira lahko predpišemo tudi v obliki UNC (*Universal Naming Convention*):

```
\\strežnik\pot\datoteka
```

ali URL (*Universal Resource Locator*), kot sta:

```
ftp://strežnik/datoteka
```

```
http://strežnik/datoteka
```

S tem so nam dostopni tudi vsi viri v krajevnem ali globalnem omrežju.

Lastnosti **Exclusive** in **ReadOnly**

Ker so podatki zapisani v datotekah, lahko ob navezavi nanje določimo način odpiranja. Če želimo drugim procesom preprečiti dostop do iste zbirke podatkov, nastavimo lastnost Exclusive na resnično vrednost (*true*). Če želimo preprečiti popravljanje vsebine zbirke podatkov, nastavimo lastnost ReadOnly.

Lastnosti **BOFAction** in **EOFAction**

Gradnik Data obsega štiri gumbe, ki omogočajo pomik po zapisih. Od leve proti desni ti gumbi omogočajo: pomik na prvi, predhodni, naslednji in zadnji zapis. Z lastnostjo BOFAction in EOFAction lahko določimo, kaj se pripeti, če se pomaknemo *pred* prvi oziroma *za* zadnji razpoložljivi zapis.

Pri lastnosti BOFAction lahko izberemo, da se podatkovni kazalec pri premiku pred prvi zapis poskusi postaviti na prvi zapis (0 – Move First) ali da ostane v nedoločenem stanju, označenem s kratico BOF (*Beginning of File*, 1 – BOF).

Za lastnost EOFAction, ki določa dogodek ob premiku za zadnji veljavni zapis, je izbira večja. Lahko izberemo pomik na zadnji zapis (0 – Move Last), obstanemo na neveljavnem zapisu, označenem s kratico EOF (*End of File*, 1 – EOF), ali pa ob tem dogodku preidemo v stanje dodajanja novega zapisa (2 – Add New).

Nedoločena stanja in celoten postopek dodajanja zapisa bomo natančneje spoznali v nadaljevanju.

Lastnost **RecordsetType**

Podatki, na katere se navežemo z lastnostjo RecordSource, lahko predstavljajo celotno tabelo ali pa so izid poizvedbe. V visual basicu lahko te podatke obravnavamo v obliki treh različnih predstavitev, imenovanih *zbirka zapisov* (*recordset*), ki se razlikujejo po namenu in učinkovitosti. Naštete so v spodnji tabeli:

Predstavitev	Uporabna za	Opis
0 – Table	Tabelo	Omogoča pregled, dodajanje, spreminjanje, brisanje, hitro preiskovanje[44].
1 – Dynaset (privzeta)	Tabelo, poizvedbo.	Omogoča pregled, dodajanje, spreminjanje, brisanje, zaporedno preiskovanje. Najmanj učinkovita.
2 – Snapshot	Tabelo, poizvedbo.	Omogoča le pregled in zaporedno preiskovanje, vendar najbolj učinkovita.

Učinkovitost posamezne predstavitve predstavlja njeno hitrost in porabo sistemskih sredstev. Če potrebujemo samo pregledovanje podatkov, je mnogo bolj učinkovito, da za predstavitev uporabimo predstavitev Snapshot.

> **NAMIG** Pri neposrednem delu s podatkovnimi viri ODBC je na voljo še ena oblika zbirke, ki jo bomo spoznali pozneje.

Lastnost **Options**

Lastnost Options nastavimo na celo število, ki je lahko kombinacija nekaj vgrajenih konstant. Njihov pomen dodatno opredeli način rabe podatkov, na katere se navezuje gradnik Data. Konstante in njihov pomen so razložene v spodnji tabeli:

Konstanta	Opis
dbDenyWrite	Prepovemo spreminjanje zapisov za druge procese. Koristno v večuporabniških okoljih, npr. pri krajevnem omrežju.
dbDenyRead	Prepovemo branje zapisov iz tabele. Ta nastavitev je mogoča le za podatkovno predstavitev vrste Table.
dbReadOnly	Zapisov ni mogoče spreminjati. Podobno nastavljeni lastnosti ReadOnly.

[44] Če je za tabelo določen kak indeks.

Konstanta	Opis
dbAppendOnly	Omogočeno je samo dodajanje zapisov, ni pa mogoče brati obstoječih.
dbInconsistent	Dopuščeni so tudi popravki vsebine, ki krnijo *celovitost* zbirke podatkov.
dbConsistent	Popravki so mogoči le pri vrednostih, ki ne krnijo celovitosti zbirke. To je tudi privzeta nastavitev.
dbSQLPassThrough	Prepusti obdelavo stavka SQL podatkovnemu strežniku, ki ga predstavlja ustrezen vir ODBC.
dbSeeChanges	Če še kak uporabnik spreminja podatke, ki jih obdelujemo, se sproži izvajalna napaka.

Več konstantnih vrednosti lahko združimo s seštevanjem ali bitnim operatorjem Or.

> **NAMIG** Na voljo je še ena konstanta, dbForwardOnly, ki določa posebej učinkovito obliko predstavitve podatkov, v kateri je dovoljen samo pomik naprej (*forward-only scrolling*). Uporabljamo jo lahko le pri predmetih DAO, pri gradniku Data pa je prepovedana.

Lastnosti **DefaultType** in **DefaultCursorType**

Pri virih ODBC se lahko izognemo podatkovnemu stroju Jet in komuniciramo neposredno z gonilnikom podatkovnega vira. V tem primeru lahko določimo, kdo vzdržuje podatkovni kazalec. Več o neposrednem delu z viri ODBC bomo spoznali v nadaljevanju.

Tako, sprehodili smo se po pomembnejših lastnostih gradnika Data, ki so na voljo tako med načrtovanjem kakor v programski kodi. Za pisanje programske kode pa je izrednega pomena lastnost, vidna le med izvajanjem. To je...

Lastnost **Recordset**

Lastnost Recordset je predmet, ki natančneje opisuje podatke, določene z lastnostjo RecordSource (izbrano zbirko zapisov). Njegove lastnosti in metode omogočajo obdelovanje podatkov iz programske kode. Natančneje ga bomo spoznali v nadaljevanju.

Oglejmo si še pomembnejše lastnosti vezanih gradnikov. Omenili smo že, da vezani gradnik prepoznamo po tem, da ima lastnost DataSource.

Lastnost **DataSource**

Ta določa ime ustreznega gradnika Data, ki bo priskrbel podatke. Pri nastavljanju te lastnosti pomaga seznam, ki vključuje imena vseh nameščenih gradnikov Data. Lastnost *moramo* nastaviti med načrtovanjem, ker iz programske kode ni dostopna.

Lastnost **DataField**

Lastnost DataField pri vezanih gradnikih določa polje, katerega vsebino bo gradnik prikazoval. Za razliko od lastnosti DataSource je lastnost DataField mogoče določiti tako med načrtovanjem kakor iz programske kode.

Lastnost **DataChanged**

Vezani gradniki poznajo še lastnost DataChanged, ki je vidna le iz programske kode. Njena vrednost je logična konstanta, ki določa, ali se je vsebina gradnika naknadno spremenila po uskladitvi s trenutnim zapisom. Sama uskladitev vsebine z zapisom (ob premiku ali klicu nekaterih metod) to lastnost nastavi na neresnično vrednost, morebitne poznejše spremembe pa jo postavijo na vrednost logične resnice.

Metode

Gradnik Data ne pozna posebnih metod za neposredno delo s podatki. Temu so namenjene lastnosti in metode njemu podrejenega predmeta Recordset. Kljub temu pozna nekaj pomembnih metod, ki vplivajo na usklajenost vezanih gradnikov in podatkov v zbirki.

Metoda **Refresh**

Metodo Refresh že poznamo. Z njo lahko vsilimo obnovitev podobe gradnika. Pri gradniku Data ima ta metoda še poseben pomen. Če spremenimo katerokoli od lastnosti, ki določajo podatkovni vir oziroma njegovo vedenje (lastnosti Connect, DatabaseName, ReadOnly, Exclusive in Options), bo klic metode Refresh zaprl in spet odprl zbirko podatkov ter tako osvežil stanje lastnosti Recordset.

Metodi **UpdateControls** in **UpdateRecord**

Metoda UpdateControls obnovi vsebino vezanih gradnikov glede na trenutni zapis v izbrani zbirki zapisov (ta je določena z lastnostjo RecordSource, predstavlja pa jo lastnost Recordset). Ravno nasprotno doseže metoda UpdateRecord, ki trajno shrani vnose v vezanih gradnikih. Podoben učinek smo preizkusili v prvem zgledu rabe vezanih gradnikov, le da je tam trajni zapis povzročil premik na nov zapis.

Dogodki

Dogodki gradnika Data odražajo spremembe, ki se dogajajo kjerkoli znotraj celotne zbirke zapisov, ki jo predstavlja. Odzivne procedure teh dogodkov so ključ za razvoj učinkovitega programa.

Dogodek **Validate**

Dogodek Validate se zgodi *pred* vsako ključno operacijo, ki naj bi se izvršila nad zbirko zapisov. V njegovi odzivni proceduri imamo možnost operacijo preklicati, spremeniti njen vpliv ali pa jo nespremenjeno prepustiti naprej.

Dogodek spremljata dva parametra, ki ju lahko znotraj procedure spremenimo in s tem vplivamo na nadaljnji potek izvajanja operacije. Prvi parameter je Action, ki opisuje vrsto operacije. Možne vrednosti tega parametra so:

Konstanta	Opis
vbDataActionMoveFirst	Premik na prvi zapis.
vbDataActionMovePrevious	Premik na prejšnji zapis.
vbDataActionMoveNext	Premik na naslednji zapis..
vbDataActionMoveLast	Premik na zadnji zapis.
vbDataActionAddNew	Vstop v stanje dodajanja zapisa.
vbDataActionUpdate	Shranjevanje vnesenih sprememb.
vbDataActionDelete	Brisanje zapisa.
vbDataActionFind	Zaporedno iskanje.
vbDataActionBookmark	Označitev trenutnega položaja (*Bookmark*).
vbDataActionClose	Zapiranje seznama zapisov.
vbDataActionUnload	Odstranitev obrazca, ki vsebuje gradnik.
vbDataActionCancel	Prekliče operacijo. Ta vrednost se ne bo nikoli posredovala, lahko pa jo nastavimo znotraj odzivne procedure in s tem preprečimo nadaljnje izvajanje začete operacije.

Drug vplivni parameter je Save. Čeprav je določen kot podatkovni tip Integer, je dejansko le v dveh stanjih, ki ju ponazarjata logični konstanti True in False[45]. Parameter določa, ali se je spremenila vsebina v vezanih gradnikih. Če želimo, ga lahko postavimo na neresnično vrednost in s tem preprečimo kakršnokoli posodabljanje vsebine.

Oglejmo si dva zgleda. V prvem preprečimo kakršnokoli trajno spremembo:

```
Private Sub Data1_Validate(Action As Integer, Save As Integer)
  Save = False
End Sub
```

[45] Vzrok za to je v združljivosti s staro kodo. Tudi tu velja, da nična vrednost predstavlja neresico, druge pa resnico.

Drugi zgled onemogoči vse operacije, razen premika naprej, v katerega preslika vse druge premike in tudi poizkus dodajanja zapisa:

```
Private Sub Data1_Validate(Action As Integer, Save As Integer)
  Select Case Action
    Case vbDataActionMoveFirst, vbDataActionMoveLast, _
         vbDataActionMoveNext, vbDataActionMovePrevious, vbDataActionAddNew
      Action = vbDataActionMoveNext
    Case vbDataActionUnload
      ' lahko preprečimo celo odstranitev obrazca
      If MsgBox("Resnično končam?", vbYesNo + vbQuestion) = vbNo Then
        Action = vbDataActionCancel
      End If
    Case Else
      Action = vbDataActionCancel
  End Select
End Sub
```

Dogodek **Error**

Po vpeljavi dogodka Error je marsikateri programer v visual basicu laže zadihal. V starejših različicah visual basica je bilo namreč mogoče podreti vsak program, ki je uporabljal zbirko podatkov in omogočal npr. brisanje zapisov do izpraznitve. Zadoščalo je, da je kateri izmed vezanih gradnikov kazal na neveljavni zapis, in program se je neslavno podrl z zloglasnim sporočilom »No current record.« Čeprav dostop do neveljavnega zapisa povzroči ulovljivo izvajalno napako, je bil problem hud. Kako jo ujeti, ko pa se zgodi nekje zunaj dosega naše kode? Kot vemo, je mogoče prestrezovalnike napak postaviti le znotraj posameznih podprogramov. Kaj pa, če se napaka pripeti tedaj, ko ni dejaven nobeden izmed njih?

No, sedaj se takrat pokliče odzivna procedura dogodka Error. Spremljata jo dva parametra. Prvi, imenovan DataErr, je koda napake, kot jo poznamo že iz obravnave predmeta Err. Seznam vseh kod napak najdemo v sprotni pomoči. Drugi parameter, Response, vpliva na nadaljnji potek izvajanja. Nastavimo ga lahko na dve konstantni vrednosti z naslednjim pomenom:

Konstanta za parameter Response	Pomen
vbDataErrContinue	Nadaljuj izvajanje.
vbDataErrDisplay	Prikaži sporočilo o napaki (privzeta nastavitev).

NAMIG Dogodek se ne bo sprožil za napake, ki se pripetijo pred dogodkom Load. Te so lahko posledica napačnih nastavitev lastnosti in jih moramo odpraviti med načrtovanjem.

Dogodek **Reposition**

Dogodek Validate se pripeti pred izvršitvijo operacije in zato lahko vpliva nanjo. Dogodek Reposition pa se pripeti *po* izvršeni operaciji, ki kakorkoli spremeni podatkovni kazalec. Odziv na ta dogodek nam omogoča uskladitev nevezanih gradnikov.

Oglejmo si nadgradnjo prvega zgleda. Denimo, da imamo na obrazcu še eno oznako, imenujmo jo Label3, v kateri bi radi prikazali starost knjižnega naslova:

```
Private Sub Data1_Reposition()
  Label3.Caption = "Izdano pred: " & _
    Year(Now) - Data1.Recordset.Fields("Year Published") & _
    " leti"
End Sub
```

Mimogrede smo še nakazali, kako iz programske kode pridemo do podatkov. Lastnosti in metode predmeta Recordset so pravkar na vrsti.

> **NAMIG** Z odzivno proceduro dogodka Validate lahko vplivamo na vedenje dogodka Reposition. Če bi uporabili zadnja dva zapisana zgleda skupaj, bi se dogodek Reposition pripetil le pri premiku na naslednji zapis, pri drugih premikih – pretvarjamo jih znotraj odziva na dogodek Validate – pa ne.

Uporaba predmeta Recordset

Gradnik Data predstavlja zbirko zapisov kot celoto. Za delo s posameznimi zapisi je na voljo njegova lastnost Recordset, ki je znova predmet s svojimi lastnostmi in metodami, dostopen le iz programske kode. Na voljo imamo metode za premik po zapisih, zaporedno in hitro iskanje ter popravljanje, brisanje in dodajanje zapisov. Namesto ločenega opisa njegovih lastnosti in metod bomo spoznali vsa pomembnejša opravila, ki jih lahko opravimo z zbirko zapisov, ki jo predstavlja.

Analiza položaja

Ker imamo pri uporabi zbirk podatkov v določenem trenutku na voljo le en sam zapis, je zelo pomembno, da poznamo njegovo veljavnost in položaj znotraj zbirke zapisov, ki mu pripada. Za ugotavljanje in določanje položaja imamo na voljo več lastnosti:

Lastnost	Opis
BOF	Označuje neustrezen položaj kazalca pred prvim veljavnim zapisom.

Lastnost	Opis
EOF	Označuje neustrezen položaj kazalca za zadnjim veljavnim zapisom.
Bookmarkable	Lastnost določa, ali je vsakemu zapisu v zbirki mogoče prirediti enolično *oznako*. Ta oznaka nam rabi za označevanje trenutnega položaja (glej spodaj).
Bookmark	Vrne ali določi enolično oznako zapisa in s tem tudi dejaven zapis. Na voljo le pri zbirkah zapisov, ki imajo nastavljeno lastnost Bookmarkable.
RecordCount	Lastnost vrne število veljavnih zapisov, do katerih smo že imeli *dostop*. To ni nujno število vseh zapisov v zbirki! Lastnost RecordCount bo podala skupno število zapisov v zbirki le, če smo že prišli do zadnjega zapisa in nihče drug hkrati z nami nima dostopa do istih podatkov. Premik na zadnji zapis sicer zahteva le klic metode, vendar lahko občutno zmanjša učinkovitost zbirke (beri: hitrost dela). Premik na zadnji zapis izsilimo le, če res nujno potrebujemo skupno število zapisov v zbirki.
AbsolutePosition	Ta lastnost je na voljo pri zbirkah vrste Snapshot in Dynaset. Vrne ali določi zaporedno številko dejavnega zapisa (tip Long), pri čemer ima prvi zapis zaporedno številko 0, število –1 pa označuje, da ni dejavnega zapisa (podobno kakor npr. lastnost ListIndex pri seznamih). Za učinkovito premikanje na že obiskane zapise ta lastnost ni uporabna in moramo še vedno poseči po lastnosti Bookmark. Poleg tega bo postavljanje lastnosti AbsolutePosition na vrednost še neobiskanega zapisa povzročila izvajalno napako. Za učinkovito rabo te lastnosti se moramo najprej pomakniti na zadnji zapis (glej zgoraj).
PercentPosition	Določa ali nastavi približen položaj dejavnega zapisa znotraj zbirke kot odstotek celotnega števila zapisov v intervalu 0,00–100,00 (tip Single). To lastnost lahko uporabimo v kombinaciji z drsnikom za hitro, a manj natančno pomikanje po zbirki zapisov.

Vidimo, da je ugotavljanje skupnega števila zapisov in absolutnega položaja lahko precej zapleteno. Toda vprašati se moramo: kdaj res nujno potrebujem skupno število zapisov in absolutni položaj dejavnega zapisa? In kako naj te vrednosti smiselno uporabim v večuporabniških okoljih, v katerih lahko več uporabnikov *hkrati* dodaja in odstranjuje zapise? V praksi se izkaže, da tako ali tako delamo le s približnimi vrednostmi. Če pa pridemo do primera, ko je absoluten položaj res ključnega pomena, lahko z zaklepanjem (*locking*) in poselitvijo zbirke (*recordset population*) zagotovimo natančnost lastnosti RecordCount in AbsolutePosition.

Oglejmo si še zgled (zgled predpostavlja, da so na vmesniku gradnik Data, oznaka ter dva drsnika, vsi s privzetimi imeni. Poleg tega mora imeti oznaka postavljeno lastnost DataSource):

```
' Osnovna vzpostavitev gradnikov
Private Sub Form_Load()
  ' Vzpostavimo gradnike
  Data1.DatabaseName = "Biblio.MDB"
  Data1.RecordSource = "Titles"
  ' med načrtovanjem: Label1.DataSource = Data1
  Label1.DataField = "Title"
End Sub

' Uporabimo predmet Recordset za vzpostavitev drsnikov
Private Sub Form_Activate()
  ' Poselimo zbirko
  With Data1.Recordset
    .MoveLast
    .MoveFirst
  End With

  ' Določimo obseg drsnikov
  With HScroll1
    .Min = 0
    .Max = 100
    .LargeChange = 10
  End With
  With HScroll2
    .Min = 0
    .Max = Data1.Recordset.RecordCount
    .LargeChange = .Max \ 10
  End With
End Sub

' Uskladimo drsnike
Private Sub Data1_Reposition()
  HScroll1.Value = Data1.Recordset.PercentPosition
  HScroll2.Value = Data1.Recordset.AbsolutePosition
End Sub

' Spremenimo položaj
Private Sub HScroll1_Change()
  HScroll1_Scroll
End Sub

Private Sub HScroll1_Scroll()
  Data1.Recordset.PercentPosition = HScroll1.Value
End Sub

Private Sub HScroll2_Change()
  HScroll2_Scroll
End Sub

Private Sub HScroll2_Scroll()
  Data1.Recordset.AbsolutePosition = HScroll2.Value
End Sub
```

Dostop do predmeta Recordset znotraj dogodka Load še ni mogoč, saj se tedaj predmet šele vzpostavlja. Vzpostavitev drsnikov zato opravimo znotraj dogodka Activate. Prvi korak pred uporabo lastnosti za opis položaja je poselitev zbirke (*recordset population*). To dosežemo s klicem metod za pomik na zadnji in nazaj na prvi zapis. S tem se učinkovitost zbirke resda zmanjša, vendar je za delovanje našega zgleda to kritična operacija. Preostanek zgleda usklajuje spremembe med gradniki.

Premiki

Ugotovili smo, da so premiki v programski kodi včasih nujni. Oglejmo si vse metode za pomik po zbirki zapisov. Z različnimi metodami Move se lahko pomikamo po zapisih enako, kakor omogočajo štirje gumbi na gradniku Data, na voljo pa je še ena, zmogljivejša oblika. Metode in njihov pomen so razloženi v spodnji tabeli:

Metoda	Opis
MoveFirst	Premakne podatkovni kazalec na prvi *veljavni* zapis.
MovePrevious	Premakne kazalec na prejšnji zapis. Lahko ga premakne tudi pred prvi zapis, ki ga ponazarja neveljavno stanje BOF[46].
MoveNext	Premakne kazalec na naslednji zapis. Lahko ga premakne tudi za zadnji zapis, ki ga ponazarja neveljavno stanje EOF[46].
MoveLast	Premakne kazalec na zadnji *veljavni* zapis.
Move	Premakne kazalec za poljubno število zapisov.

Metoda Move prejme še dva parametra. Njena popolna oblika se glasi:

```
Move odmik [, oznaka]
```

Odmik podaja relativni odmik od trenutnega zapisa. Če gre za pozitivno število, se bomo pomaknili naprej, negativno število pa predstavlja pomik nazaj v zbirki zapisov. Po želji lahko predpišemo še drugi parameter, ki mora biti oznaka zapisa, shranjena v lastnosti Bookmark. V tem primeru prvi parameter določa relativni odmik od shranjenega in ne več od trenutnega zapisa.

Značilen zgled rabe metod je sprehod po vseh razpoložljivih zapisih. Spodnji zgled si zapomni trenutni položaj, zatem se sprehodi po zapisih od zadnjega proti prvemu, na koncu pa se vrne na izhodišče:

```
Dim vntOznaka

With Data1.Recordset
```

[46] Na delovanje metod *ne* vplivata nastavitvi lastnosti BOF/EOFAction gradnika Data.

```
  ' Shranimo položaj, če je mogoče
  If .Bookmarkable Then vntOznaka = .Bookmark

  .MoveLast
  Do Until .AbsolutePosition = 0

    ' operacija nad zapisom...

    ' omogočimo izvajanje drugih opravil
    DoEvents

    .MovePrevious

  Loop

  ' Obnovimo morebiti shranjen položaj
  If Not IsEmpty(vntOznaka) Then .Bookmark = vntOznaka
End With
```

Ponavadi v glavi zanke preverjamo stanje BOF ali EOF. Mi smo uporabili lastnost AbsolutePosition. S tem se izognemo premiku na neveljaven zapis, ki bi lahko povzročil izvajalno napako, npr. ko uporabljamo odziv na dogodek Reposition. Kot vemo, se ta dogodek pripeti po vsakem premiku, najsi ga povzroči uporabnik ali programska koda.

Neveljaven zapis

Pri delu z zbirkami zapisov lahko podatkovni kazalec v določenih primerih kaže na neveljaven zapis. To se lahko zgodi v več primerih:

- Premaknili smo se pred prvi ali za zadnji zapis v zbirki.
- Dejavni zapis je odstranjen.
- Zbirka zapisov je povsem prazna.

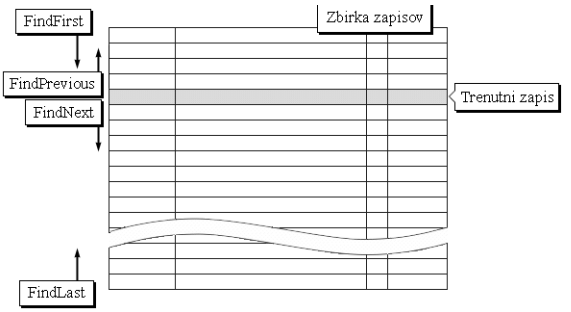
Večino teh stanj natančno opisujeta prej omenjeni lastnosti BOF in EOF. Oglejmo si pomen vseh različnih kombinacij, ki ju lahko določata:

Lastnost BOF	Lastnost EOF	Pomen
False	False	Veljaven zapis, razen če ni bil ravnokar odstranjen.
False	True	Neveljaven zapis za zadnjim zapisom v zbirki.
True	False	Neveljaven zapis pred prvim zapisom v zbirki.
True	True	Zbirka zapisov je prazna. Neveljaven zapis.

Iskanja

V vsaki zbirki zapisov lahko z zaporednim preiskovanjem najdemo vse zapise, ki ustrezajo poljubnemu merilu. Pri tem nam pomagajo naslednje metode:

Metoda	Opis
FindFirst	Začne iskati pri vrhu in najde prvi zapis, ki ustreza merilu.
FindNext	Najde naslednji zapis, ki ustreza merilu.
FindPrevious	Najde predhodni zapis, ki ustreza merilu.
FindLast	Začne iskati pri dnu in najde zadnji zapis, ki ustreza merilu.



Slika 11-5: Potek različnih metod iskanja

Vse metode prejmejo parameter, ki je v nizu znakov zapisano merilo v obliki logičnega izraza. Uspešnost zadnjega iskanja določa lastnost **NoMatch**. Oglejmo si zgled, ki najde vse zapise, ustrezne merilu:

```
Dim strKriterij

strKriterij = "Len(Title) > 10 And Title Like '*visual basic*'"

With Data1.Recordset

  .FindFirst strKriterij

  Do Until .NoMatch

    If MsgBox("Najden zapis. Nadaljujem?", vbYesNo) = vbNo Then
      Exit Do
    End If

    .FindNext strKriterij

  Loop

End With
```

V nizu, ki določa merilo, lahko uporabimo vse vgrajene funkcije in operatorje visual basica. Prav gotovo je zelo uporabna primerjava z operatorjem Like, ki ni občutljiva za velikost znakov.

> **NAMIG** Pogoj, ki ga podamo metodam Find, je enakovreden izrazu, ki sledi določilu WHERE v stavkih SQL.

Samo v tabelah z vzpostavljenim indeksom, s katerimi se ukvarja podatkovni stroj Jet, je mogoče hitro preiskovati z metodo **Seek**. Metoda Seek prejme od dva do štirinajst parametrov, ki določajo način primerjave in iskani ključ. Pester nabor parametrov izvira iz dejstva, da je iskanje z metodo Seek navezano na vzpostavljeni indeks. Ker ima lahko indeks do trinajst ključev, je tolikšno tudi število dodatnih parametrov metode. Najprej pa si oglejmo različne možnosti za primerjavo, ki jo določa prvi parameter:

Vrednost parametra primerjave	Opis
<	Začne preiskovati na koncu urejenega zaporedja zapisov in najde prvi zapis, ki ne ustreza več podanim ključem (gledano od spodaj).
<=	Najde prvi ustrezni zapis ali prvega, ki sledi, če noben ne izpolnjuje podanih ključev (gledano od spodaj).
=	Začne preiskovati pri vrhu in najde prvi zapis, ki popolnoma ustreza podanim ključem. Učinkovita raba te primerjave je smiselna le, če so podani prav vsi ključi, določeni v indeksu.
>=	Najde prvi ustrezni zapis ali prvega, ki sledi, če noben ne izpolnjuje podanih ključev (gledano od zgoraj).
>	Najde prvi zapis, ki ne ustreza več podanim ključem (gledano od zgoraj).

Parameter primerjave vedno zapišemo znotraj niza. Oglejmo si zgled, ki uporablja priloženo zbirko Biblio.MDB:

```
With Data1

  ' določimo zbirko zapisov vrste Table
  .RecordsetType = vbRSTypeTable
  .RecordSource = "Titles"
  .Refresh

  With .Recordset

    .Index = "Title"

    .Seek ">=", "The Database Book"
    If Not .NoMatch Then MsgBox "Knjiga je najdena"

    .Seek ">", "The Database Book"
    If Not .NoMatch Then MsgBox "Prva, ki sledi"

    .Seek "<", "M"
    If Not .NoMatch Then MsgBox "Zadnja z naslovom na L"

  End With

End With
```

Uspešnost iskanja znova predstavlja lastnost NoMatch. Za uspešno uporabo hitrega iskanja moramo natančno poznati imena in zgradbo indeksov. Njihov pregled, pa tudi obdelava je mogoča s programsko kodo, kot bomo spoznali pri delu s predmeti DAO. Pri izključni rabi gradnika Data moramo bodisi natančno poznati notranjo zgradbo zbirke podatkov, bodisi posežemo po posebnem orodju Visual Data Manager, ki je priloženo visual basicu. Njegov opis bomo spoznali malo pozneje.

Dostop do vsebine

Vezani gradniki samodejno prikazujejo vsebino polj dejavnega zapisa. Med vezanimi gradniki najdemo celo gradnika PictureBox in Image, s katerima lahko neposredno pregledujemo tudi slike, shranjene v zbirki.

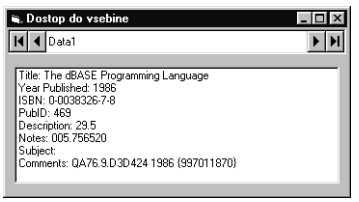
Če bi prikaz vrednosti radi nadzorovali programsko, npr. pri rabi nevezanih gradnikov, izračunanih vrednosti ali za prikaz posebnih vsebin, lahko do podatkov pridemo tudi neposredno. Pri tem nam pomaga zbirka **Fields**, ki je lastnost predmeta Recordset. Še enkrat si oglejmo osnovne značilnosti zbirke (*collection*): gre za polje elementov, ki ga lahko naslavljamo bodisi z zaporedno številko bodisi z opisnim imenom, sprehod po elementih zbirke pa je mogoč z zanko For Each.

Posamezen element zbirke Fields je predmet **Field**, ki predstavlja posamezno polje zapisa. Ima spet svoje lastnosti in metode, vsebuje celo novo zbirko. V praksi jih uporabljamo le peščico. Podrobnejši opis predmeta Field najdemo v sprotni pomoči.

Oglejmo si zgled, ki izpiše osnovne podatke o poljih:

```
Private Sub Data1_Reposition()
  Dim f As Field

  List1.Clear
  For Each f In Data1.Recordset.Fields
    List1.AddItem f.Name & ": " & f.Value
  Next
End Sub
```

Ob vsakem premiku se seznam zapolni z imeni in vsebino vseh polj dejavnega zapisa, kar vidimo na spodnji sliki:



Slika 11-6: Pregled vseh polj zapisa

Privzeta lastnost

Zaradi močno razvejene predmetne zgradbe lahko pride do nadvse dolgih sklicevanj. Ta sklicevanja niso le dolgovezna, temveč povzročijo tudi manjšo učinkovitost programske kode, saj vsaka uporaba operatorja sklicevanja (pika, .) pomeni nekaj dodatnega dela za izvajalni sklop visual basica. Večina predmetov zato pozna *privzeto lastnost*. Na privzeto lastnost (*default value*) se lahko sklicujemo samo z zapisom imena predmeta. Pri predmetu Field je privzeta lastnost Value, zato sta naslednja dva stavka enakovredna:

```
MsgBox Data1.Recordset.Fields("Title").Value
MsgBox Data1.Recordset.Fields("Title")
```

Pa ne samo to, za predmet Recordset je privzeta lastnost zbirka Fields. Kadar je privzeta lastnost predmeta zbirka, moramo za krajši zapis uporabiti poseben operator sklicevanja, klicaj (!, *bang operator*), ki mu sledi opisna oznaka elementa zbirke. Zgornjima stavkoma je zato enakovreden tudi tale:

```
MsgBox Data1.Recordset!Title
```

Če bi opisna oznaka slučajno vključevala presledek, bi jo morali objeti med oglata oklepaja:

```
MsgBox Data1.Recordset![Year Published]
```

> **NAMIG** Privzete lastnosti poznajo tudi gradniki uporabniškega vmesnika. Za oznako npr. je privzeta lastnost Caption. Tega pravila pri gradnikih nismo omenjali, ker njegova raba precej zmanjša berljivost programske kode, na učinkovitost pa praktično ne vpliva. Če želite, si lahko privzete lastnosti gradnikov ogledate v sprotni pomoči.

Popravljanje in dodajanje vsebine

Vsebino zapisov je mogoče popravljati brez posebnih metod. Če se spremeni vsebina vezanega gradnika in se prestavimo na nov zapis, bodo spremembe shranjene, razen če shranjevanja ne preprečimo z ustrezno odzivno proceduro dogodka Validate. Če želimo, lahko potek popravljanja vrednosti od začetka do konca nadzorujemo izključno iz programske kode. To je še posebej pomembno pri popravljanju podatkov, ki morajo ustrezati določenim pogojem.

Ali je popravljanje zapisov za določeno zbirko sploh mogoče, nam pove lastnost **Updatable**. Če je lastnost resnična, to še ne pomeni, da je mogoče popraviti vsebino *vsakega* polja. Notranja pravila v zbirki podatkov lahko preprečujejo posodabljanje določenih polj. Predmet Field zato pozna lastnost **DataUpdatable**, ki določa možnost posodabljanja vsebine posameznega polja.

Pri popravljanju ali dodajanju vsebine naletimo na *vmesno kopijo* (*copy buffer*). Vrednosti, ki so prikazane v vezanih gradnikih z možnostjo vnosa, so pravzaprav vmesna kopija podatkov iz zapisa. Ta kopija se samodejno ustvari, kadarkoli želimo popraviti obstoječe podatke ali vnesti nov zapis. Spremembe se zato ne shranijo, dokler jih izrecno ne potrdimo. Ob potrditvi se vsebina vmesne kopije prepiše v ustrezen zapis, sicer pa se spremembe ne shranijo.

Pri vezanih gradnikih, ki omogočajo popravljanje vsebine, kot je npr. vnosno polje, se vmesna kopija samodejno ustvari pri prikazu vsebine in tudi samodejno shrani ob prehodu na nov zapis, če tega ne prepreči dodatna programska koda. Pri popolnoma programskem nadzoru dodajanja pa se vmesna kopija pripravi pri klicu metod **Edit** in **AddNew**. Po klicu ene izmed metod popravimo vsebino polj, vendar s tem ne vplivamo na zapise v zbirki podatkov, saj dejansko delamo z vmesno kopijo. Na koncu moramo spremembe bodisi potrditi bodisi zavreči. Potrditev dosežemo z eksplicitnim klicem metode **Update**. Preklic je lahko posledica vnovičnega klica metode Edit ali AddNew oziroma premika na nov zapis, lahko pa spremembo prekličemo tudi eksplicitno, s klicem metode **CancelUpdate**.

Oglejmo si preprost zgled popravljanja in dodajanja vsebine, ki ne uporablja vezanih gradnikov. Popravljanje vsebine je dokaj preprosto opravilo:

```
' Popravljanje zapisa
Private Sub Command1_Click()
Dim strNaslov As String

  With Data1.Recordset

    ' Sploh lahko popravljamo vsebino?
    If Not .Updatable Then Exit Sub

    strNaslov = InputBox("Nov naslov", "Popravek", .Fields("Title"))

    If strNaslov <> "" Then
      .Edit
      !Title = strNaslov
      .Update
    End If

  End With

End Sub
```

Za dostop do vsebine smo uporabili okrajšani zapis z operatorjem klicaj (!). Če je uporabnik vnesel popravek, se najprej ustvari vmesna kopija, njena vsebina se popravi, spremembe pa se trajno zapišejo šele po klicu metode Update. Precej podobno poteka tudi dodajanje novega zapisa, le da moramo tu upoštevati še celovitost podatkov:

```
' Dodajanje zapisa
Private Sub Command2_Click()
Dim strNaslov As String
Static lngISBN As Long
```

```
  ' Poskrbimo za enolično vrednost
  If lngISBN = 0 Then lngISBN = Timer

  With Data1.Recordset

    ' Sploh lahko dodajamo zapise?
    If Not .Updatable Then Exit Sub

    strNaslov = InputBox("Nov naslov", "Vnos", "")

    If strNaslov <> "" Then
      .AddNew
      !Title = strNaslov
      lngISBN = lngISBN + 1
      !ISBN = lngISBN
      !PubID = 1
      .Update
      .Bookmark = .LastModified
    End If

  End With
End Sub
```

Zaradi pravil, ki so določena v zbirki podatkov, moramo vsakemu zapisu v tabeli Titles določiti oznako založnika PubID in *enolično* vrednost knjižne številke ISBN[47]. Zato ob prvem vstopu v proceduro vzpostavimo vrednost statične spremenljivke lngISBN kar na število pretečenih milisekund od polnoči do tega trenutka (to vrednost vrača funkcija **Timer**). Seveda gre le za demonstracijo, ki bi lahko imela resne težave, če bi nekdo vnašal podatke vsak dan ob natanko isti uri.

Dodajanje zapisa lahko napravimo prijaznejše z rabo lastnosti **LastModified**. Ta lastnost vsebuje oznako (*bookmark*) zadnjega popravljenega ali dodanega zapisa. V zgornjem zgledu smo lastnost izkoristili za to, da smo se takoj po dodajanju zapisa prestavili na njegovo mesto. To se sicer ne zgodi samodejno, kajti vsak nov zapis se doda na *konec* zbirke.

Metodo, ki je povzročila stvaritev vmesne kopije podatkov, lahko preverimo z lastnostjo **EditMode**, ki ima lahko naslednje vrednosti:

Vrednost lastnosti EditMode	Pomen
dbEditNone	Ni vmesne kopije, navadno pregledovanje.
dbEditInProgress	Klic metode Edit je povzročil nastanek vmesne kopije. Spremembe še niso shranjene.
dbEditAdd	Klic metode AddNew je povzročil nastanek vmesne kopije. Spremembe še niso shranjene.

[47] V resnici je knjižna številka ISBN posebej določena in zagotovo enolična za vsako knjigo na svetu.

Zaklepanje

Ozrimo se še na upoštevanje zaklepanja zapisov pri večuporabniškem delu. Če ima do istih podatkov hkrati dostop več uporabnikov, je treba poskrbeti za njihovo usklajeno delo. Nikoli ne moremo z gotovostjo predpostaviti, da ni prišlo do sočasnega dostopa do istega zapisa. Podatkovni stroj Jet zato pred kakršnimkoli popravljanjem vsebine zaklene *stran*, na kateri je trenutni zapis. Velikost strani (*page*) je vnaprej določena in lahko obsega tudi sosednje zapise. Pri domorodnem zapisu (Microsoft Access, .MDB) obsega stran 2048 zlogov.

Zaklepanje strani (*locking*) lahko poteka pesimistično ali optimistično. Pri pesimističnem zaklepanju se stran zaklene že ob klicu metode Edit oziroma AddNew. Pri optimističnem zaklepanju pa se stran ne zaklene vse do klica metode Update, ki shrani spremembe. V obeh primerih se po vnosu ali preklicu sprememb stran sprosti. Vrsto zaklepanja določa lastnost **LockEdits**, ki je po privzeti nastavitvi postavljena na logično resnico, ki pomeni pesimistično zaklepanje.

Oglejmo si zgleda dveh funkcij, ki upoštevata možnost zaklepanja zapisa, zaradi katerega lahko posodabljanje vsebine spodleti. Prva funkcija uporablja pesimistično zaklepanje:

```
Function PopraviPesimisticno(strNovNaslov As String) As Boolean

  PopraviPesimisticno = True

  With Data1.Recordset

    ' Pesimistično zaklepanje...
    .LockEdits = True
    ' ...zato že tu lovimo napake.
On Error GoTo Zaklenjeno
    .Edit
On Error GoTo 0

    ' Če smo še vedno v stanju popravljanja, ni bilo
    ' napake zaradi zaklenjenega zapisa
    If .EditMode = dbEditInProgress Then
      ' Popravimo zapis, ki je za druge zaklenjen
      !Title = strNovNaslov
      .Update
      .Bookmark = .LastModified
    Else
      ' Vnovič preberemo dejavni zapis, da povzamemo
      ' spremembe, vnesene s strani drugega uporabnika
      .Move 0
    End If

  End With

  Exit Function

Zaklenjeno:
Dim napaka As Error
```

```
  ' Izpišemo vse napake
  For Each napaka In DBEngine.Errors
    MsgBox "Napaka: " & napaka.Number & _
        vbCr & napaka.Description
  Next
  PopraviPesimisticno = False

  ' Nadaljujemo izvajanje funkcije
  Resume Next

End Function
```

V drugem zgledu pa uporabimo optimistično zaklepanje, ki lahko vpliva na uspešnost metode Update:

```
Function PopraviOptimisticno(strNovNaslov As String) As Boolean

  PopraviOptimisticno = True

  With Data1.Recordset

    ' Optimistično zaklepanje...
    .LockEdits = False
    .Edit
    !Title = strNovNaslov
    ' ...pomeni, da se napaka lahko zgodi
    ' ob poizkusu zapisa nove vsebine
On Error GoTo Zaklenjeno
    .Update
On Error GoTo 0

    ' Če nismo več v stanju popravljanja, ni bilo
    ' napake zaradi zaklenjenega zapisa
    If .EditMode = dbEditNone Then
      ' Premaknemo se na posodobljen zapis
      .Bookmark = .LastModified
    Else
      ' Prekličemo popravljanje in osvežimo zapis
      .CancelUpdate
      .Move 0
    End If

  End With

  Exit Function

Zaklenjeno:
Dim napaka As Error

  ' Izpišemo vse napake
  For Each napaka In DBEngine.Errors
    MsgBox "Napaka: " & napaka.Number & _
        vbCr & napaka.Description
  Next
  PopraviOptimisticno = False

  ' Nadaljujemo izvajanje funkcije
  Resume Next

End Function
```

V obeh zgledih smo uporabili doslej še neznani predmet **DBEngine**, ki opisuje podatkovni stroj Jet. Njegova lastnost **Errors** je zbirka, ki obsega morebitne napake, ki so se zgodile pri delu z zbirko podatkov. Zadnja napaka v tej zbirki se preslika tudi v predmet **Err**, ki ga že poznamo iz splošne obravnave izvajalnih napak.

Odstranjevanje zapisov

Dejaven zapis lahko odstranimo iz zbirke vrste Table ali Dynaset z metodo **Delete**. To dejanje pa ima kar nekaj posledic. Tudi po uspešno opravljeni metodi Delete ostane podatkovni kazalec na istem mestu, vendar sedaj kaže na neveljaven zapis. Njegova uporaba nemudoma povzroči izvajalno napako. Redni spremljevalec klica metode Delete je zato še katera od metod Move, ki premakne kazalec na veljavno mesto. Po tem premiku se na odstranjeni zapis ne moremo več postaviti.

Oglejmo si zgled »varne« funkcije odstranjevanja zapisa:

```
Function VarnoOdstrani()

  VarnoOdstrani = True

On Error GoTo Napaka:
  With Data1.Recordset
    .Delete
    .MoveLast
  End With

  Exit Function

Napaka:
  VarnoOdstrani = False

End Function
```

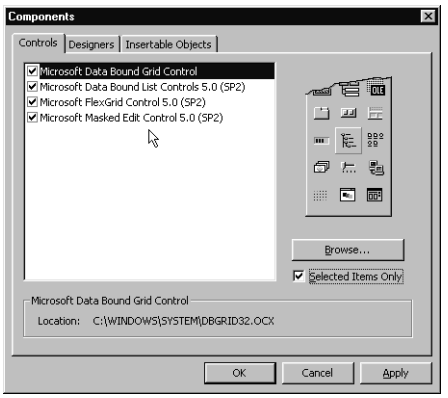
Zmogljivi vezani gradniki

Kakor smo videli, je mogoče z lastnostmi in metodami predmeta Recordset nadzorovano posnemati vse funkcije vezanih gradnikov. Zastavlja se vprašanje, kdaj je sploh smiselno uporabiti vezane gradnike? Verjetno so dobra odločitev pri preprostih projektih, ko želimo hitro rezultate in trdoživost ni najpomembnejša. Če bi radi razvili program, ki bo preverjeno deloval v vseh okoliščinah, pa je bolje poseči po programski kodi in uporabiti navadne gradnike.

Drugače je s posebnimi vezanimi gradniki, katerih zmogljivosti precej presegajo znanje in sposobnosti povprečnega programerja v visual basicu. Eden izmed njih sodi med vgrajene gradnike visual basica, drugi pa so v zunanjih knjižnicah ActiveX. To so:

- Vsebnik OLE (*OLE*). Vgrajeni gradnik, ki omogoča delo z vstavljenimi predmeti (*insertable objects*). Če ga uporabimo v vezani obliki, omogoča prikaz polj z obsežno vsebino (*large binary*).
- Vezano oblikovano vnosno polje (*MaskEdBox*). Na voljo je le v boljših različicah visual basica. Je v knjižnici Microsoft Masked Edit Control.
- Vezana seznama (*DBCombo* in *DBList*). Najdemo ju v knjižnici Microsoft Data Bound List Controls.
- Vezani preglednici FlexGrid (*MSFlexGrid*) in Data Bound Grid (*DBGrid*). Najdemo ju v knjižnicah Microsoft FlexGrid Control in Microsoft Data Bound Grid Control.

Slika 11-7: Knjižnice zunanjih vezanih gradnikov



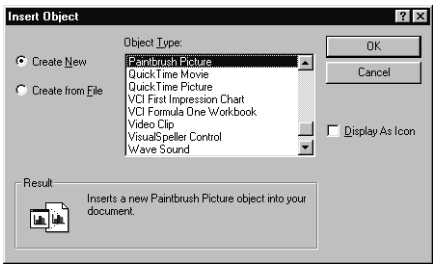
Vsi našteti gradniki so zelo uporabni tudi brez navezave na podatkovni vir.

Vstavljeni predmet

V zbirko podatkov je mogoče shraniti tudi polja z *obsežno vsebino*. Obsežna vsebina (*large binary*) je lahko kakršnakoli binarna vsebina, ki je najpogosteje vstavljivi predmet OLE. Naloga našega programa je, da to vsebino pravilno predstavi. Pri tem nam zelo pomaga vsebnik OLE, poseben vgrajeni gradnik, ki omogoča prikazovanje in vstavljeno popravljanje (*in-place editing*) povezanih ali vsebovanih predmetov.

Ko uporabljamo ta gradnik, moramo najprej določiti, kakšno vrsto predmeta bomo v njem prikazovali. Pravzaprav to od nas samodejno zahteva gradnik, ko ga dodamo uporabniškemu vmesniku. Neposredno po namestitvi gradnika se prikaže pogovorno okno, ki zahteva določitev vrste vsebovanega predmeta. Pogovorno okno je vidno na naslednji sliki:

Slika 11-8: Izbira vrste predmeta

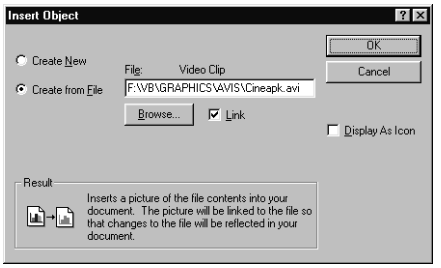


Pogovorno okno obsega seznam vseh napovedanih razredov v našem računalniku, ki podpirajo *avtomatizacijo*. Avtomatizacija (*automation*)[48] je posebna zmožnost programov, s katero pregledujemo in popravljamo njihove dokumente, ne da bi se ti programi vidno odprli. V tem primeru se dokumenti imenujejo *predmeti*, saj niso le kopica podatkov, ampak imajo tudi določeno vedenje, kar je osnovna značilnost predmetov. Avtomatizacijo predmetov podpirajo praktično vsi sodobnejši programi, omogočimo pa jo lahko tudi v posebnih programih, razvitih z visual basicom, kar bomo spoznali v četrtem delu te knjige.

Ob sprožitvi avtomatizacije se s strani uporabnika zdi, kakor da bi predmet »oživel« in ponudil možnosti za svoje posodabljanje. Natančno tako vedenje potrebujemo za prikaz in morebitno popravljanje obsežne binarne vsebine, če seveda poznamo njeno vrsto.

Pri izbiri razreda v prikazanem pogovornem oknu imamo na voljo izdelavo novega, vstavljenega izvoda (predmeta), ki ga bo podpirala avtomatizacija njegovega programa (opcijsko stikalo **Create New**) ali odpiranje obstoječega predmeta v shranjeni datoteki na disku (opcijsko stikalo **Create From File**). V tem primeru se lahko tudi odločimo, ali bo vsebina predmeta shranjena z našim programom (vstavljeni predmet, *embedded object*), ali pa bo naš program zgolj vzdrževal povezavo z datoteko (povezan predmet, *linked object*). Pri določanju povezave z datoteko seznam programov nadomesti vnosno polje za ime datoteke **File** in stikalo **Link**, ki določa morebitno povezanost namesto vstavitve.

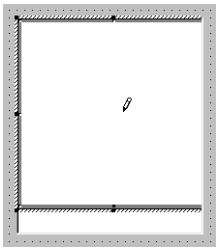
Slika 11-9: Izbira obstoječega predmeta



[48] Izraz *avtomatizacija* je nadomestil kratico OLE 2 (*Object Linking and Embedding*). Ime gradnika je ostalo zaradi združljivosti s staro kodo.

V zgledu bomo izrisanemu vsebniku OLE določili, naj vsebuje nov predmet razreda »Paintbrush Picture«. Ta razred bi moral biti na voljo praktično v vsakem računalniku z Okni. Poskrbimo, da je vklopljeno opcijsko stikalo **Create New**, in s seznama **Object Type** izberemo želeni razred (glej sliko 11-8). Ko kliknemo gumb OK, postane vsebnik OLE dejaven, saj smo mu naročili, naj vsebuje nov vstavljeni predmet. V ozadju se je nevidno pognal program, ki skrbi za avtomatizacijo predmetov razreda »Paintbrush Picture«. To lahko opazimo po posebni obrobi gradnika in spremenjenem kazalcu miške, ki ponazarja pripravljeno orodje za risanje.

Slika 11-10: Dejavni avtomatizirani predmet razreda »Paintbrush Picture«



NAMIG Če razreda »Paintbrush Picture« ne najdete na seznamu avtomatiziranih razredov, morate namestiti urejevalnik bitnih slik MS Paint, ki je sestavni, vendar neobvezni pripomoček operacijskega sistema Okna.

Če želimo, lahko na površino dejavnega predmeta kaj narišemo. V nedejavno stanje ga prestavimo tako, da izberemo kak drug gradnik na uporabniškem vmesniku. Ko dejavni predmet izgubi vnosno točko (*focus*), se avtomatizacija konča. Znova jo sprožimo tako, da v priročnem menuju vsebnika OLE izberemo ukaz **Edit**.

NAMIG V izrisani gradnik lahko vstavimo tudi predmet, shranjen na odložišču. V priročnem menuju nameščenega vsebnika OLE izberemo ukaz **Paste Special**...

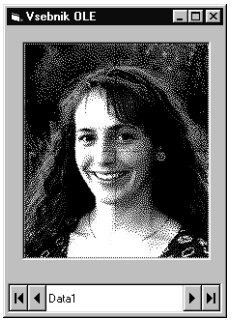
Za demonstracijo sposobnosti gradnika dodajmo uporabniškemu vmesniku še gradnik Data, ki ga bomo povezali z zbirko podatkov v datoteki Nwind.MDB. Nastaviti moramo naslednje lastnosti:

Gradnik	Lastnost	Nastavitev
podatki (*Data*)	(Name)	Data1
	DatabaseName	Nwind.MDB
	RecordSource	Employees
	Align	2 - Align Bottom

Gradnik	Lastnost	Nastavitev
vsebnik OLE (*OLE*)	(Name)	OLE1
	(pri namestitvi izberemo vstavljeni predmet razreda »Paintbrush picture«)	
	DataSource	Data1
	DataField	Photo
	SizeMode	2 - AutoSize

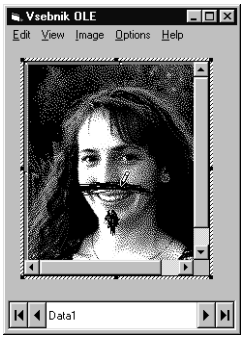
To je vse. Zgled lahko poženemo in preizkusimo.

Slika 11-11: Uporaba vsebnika OLE za prikaz obširne binarne vsebine



Znova lahko brez vrstice programske kode uživamo prednosti vezanih gradnikov. Z vsebnikom OLE nismo dobili samo možnosti pregledovanja predmetov razreda »Paintbrush Picture«, za katere sedaj vemo, da so pravzaprav bitne slike. Vsaka slika je hkrati tudi izpostavljen, avtomatiziran predmet, ki ga lahko sprožimo in s tem urejamo. Za vse bo poskrbel program MS Paint, ki bo deloval v ozadju. Predmet spravimo v dejavno stanje tako, da površino vsebnika OLE dvakrat kliknemo z miško. Dobil bo posebno obrobo, program pa menujsko vrstico programa v ozadju. Kar dobro za programček brez ene same vrstice kode.

Slika 11-12: Avtomatizacija vstavljenega predmeta



Če želimo, lahko vsebino slike popravimo s pripomočki, ki jih ponuja orodje MS Paint. Če se premaknemo na nov zapis v tabeli, bodo spremembe trajno shranjene v zbirko podatkov. Vsebnik lahko koristno uporabimo tudi pri dodajanju novih zapisov in pri rabi kateregakoli drugega vezanega gradnika.

Vsebnik OLE nam omogoča, da tudi dokumente programov, ki podpirajo avtomatizacijo, preprosto shranjujemo v zbirko podatkov in jih urejamo z ustreznimi orodji! To niso samo bitne slike, v zbirko lahko shranimo npr. preglednico iz Excela, dokument iz Worda, sliko iz Corela..., skratka katerikoli avtomatizirani predmet, ki je na voljo v našem sistemu. Če omogočimo avtomatizacijo, moramo seveda poskrbeti, da je izbrani razred avtomatiziran tudi v drugih računalnikih, v katere želimo namestiti naš program.

NAMIG Med vezanimi vgrajenimi gradniki najdemo tudi gradnika PictureBox in Image, ki sta sposobna prikazovati slike. Ker smo v prejšnjem zgledu prikazovali in popravljali vsebino polja, v katerem so shranjene *predmetno* predstavljene bitne slike, omenjenih gradnikov ni mogoče uporabiti, saj njuni lastnosti Picture ne moremo prirediti predmeta, temveč le golo bitno sliko.

Omenjena gradnika sta uporabna npr. takrat, ko dopolnjujemo še prazno tabelo. Če ji *dodamo* zapise z vrednostmi v vezanih gradnikih, bo vsebina v povezanih poljih izvirala neposredno iz gradnika PictureBox ali Image. Vsebina tako ne bo več predmet, temveč čista bitna slika. Še vedno pa bo v zbirki podatkov to polje predstavljeno kot obsežna vsebina, ki jo bomo podrobneje spoznali, ko bomo obravnavali strukturo zbirke podatkov.

Lastnosti

Pregledno si oglejmo pomembnejše lastnosti vsebnika OLE, dostopne med načrtovanjem vmesnika, ki nam lahko pomagajo bolje izkoristiti njegove zmožnosti:

Lastnost	Opis
Class	Določa razred predmeta v vsebniku OLE. Nastavitev te lastnosti je lahko bolj zapletena, saj nam visual basic prikaže prava imena razredov in ne opisnih imen, ki so na voljo v pogovornem oknu ob namestitvi vsebnika na obrazec (opisnemu imenu »Paintbrush picture« npr. pripada pravo ime Paint.Picture).
OLEDropAllowed	Lastnost določa, ali vsebnik sprejme predmete, ki so nanj odvrženi s tehniko »povleci in izpusti« (*drag and drop*).

Lastnost	Opis
OLETypeAllowed	Ta lastnost lahko omeji vrsto predmetov, ki jih vsebnik sprejme. Ti so lahko samo povezani (0 – Linked) ali samo vstavljeni (1 – Embedded). Privzeta nastavitev dopušča oboje (2 – Either).
SourceDoc	Lastnost uporabimo pri obstoječih, shranjenih predmetih. Priredimo ji ime datoteke z dokumentom.
SourceItem	Skupaj z lastnostjo SourceDoc lahko lastnost SourceItem dodatno opredeli vsebino znotraj datoteke, ki pripada vstavljenemu ali povezanemu predmetu.
AutoActivate	Določa način sprožitve vstavljenega predmeta med izvajanjem. Izbiramo lahko med ročnim aktiviranjem (0 – manual), ki ga moramo sprožiti iz programske kode, lahko se predmet sproži takoj, ko vsebnik OLE prejme vnosno točko (1 – GetFocus), privzeta nastavitev je dvakratni klik (2 – DoubleClick), lahko pa odločitev prepustimo tudi razredu predmeta (3 – Automatic). Pri nekaterih razredih je vzrok sprožitve dvakratni klik, pri drugih je lahko to že prejem vnosne točke.
AutoVerbMenu	Lastnost določa, ali se med izvajanjem samodejno prikaže priročni menu razreda, ko kliknemo vsebnik z desno tipko miške.
DisplayType	Določa ponazoritev vsebine v vsebniku. Namesto dejanske vsebine lahko vidimo samo ikono (1 – Icon). To nastavitev lahko spremenimo samo, če v vsebniku ni nobenega predmeta, kar dosežemo z izbiro **Delete Embedded Object** iz priročnega menuja vsebnika OLE.
SizeMode	Določa način prilagajanja velikosti vsebnika vsebini in nasprotno. Na voljo imamo obrezovanje (0 – Clip), raztegovanje (1 – Stretch), samodejno prilagajanje vsebini (2 – AutoSize) in povečevanje (3 – Zoom).

Pri uporabi vsebnika OLE smo ugotovili, da sproženi predmet prikaže menujsko vrstico pripadajočega programa v ozadju, ki ga nadzoruje. To značilnost lahko tudi onemogočimo. Nastaviti moramo posebno lastnost obrazca (*Form*), **NegotiateMenus**, ki jo postavimo na neresnično vrednost. Če prikaz menuja dopustimo in ima tudi naš program lastne menuje, lahko vsakemu elementu glavne menujske vrstice (*Menu*) določimo lastnost **NegotiatePosition**, ki določa način razvrščanja menuja, ko je menujska vrstica programa v ozadju.

Uporaba iz kode

V programski kodi je za uporabo vsebnika OLE najpomembnejša metoda **DoVerbs**, ki izvede katerokoli operacijo, podprto v razredu, ki mu vstavljeni predmet pripada.

Te ukaze lahko za posamezni predmet pregledamo v njegovem priročnem menuju med izvajanjem programa (če tega ne omogočimo z lastnostjo AutoVerbMenu). Razpoložljive operacije lahko pregledamo tudi programsko. Metoda **FetchVerbs** prebere razpoložljive ukaze in jih postavi na voljo skozi lastnost **ObjectVerbsCount**. Poleg podprtih operacij si lahko ogledamo tudi podprte oblike vsebine. Lastnost **ObjectAcceptFormats** določa vsebine, ki jih vsebnik lahko sprejme, lastnost **ObjectGetFormatsCount** pa vsebine, ki jih predmet lahko postavi na odložišče (*clipboard*). Za vstavitev novega predmeta v vsebnik pa lahko pokličemo metodo InsertObjDlg, ki prikaže enako pogovorno okno, kakršno smo videli, ko smo vsebnik nameščali na obrazec. Uporabo naštetih elementov ponazarja naslednji zgled, ki poleg vsebnika OLE (OLE1) potrebuje tri odprte sezname (List1, List2 in List3) in ukazni gumb (Command1):

```
Private Sub Command1_Click()
Dim i As Integer

  ' Prikažemo pogovorno okno,
  ' uporabnik naj izbere želeni razred.
  OLE1.InsertObjDlg

  ' Preberemo razpoložljive ukaze
  OLE1.FetchVerbs

  List1.Clear
  List2.Clear
  List3.Clear

  ' Zapolnimo seznam. Ker element ObjectVerbs(0) vsebuje
  ' privzeti ukaz, ki se ponovi v ObjectVerbs(1), začnemo zanko
  ' z zaporedno številko 1 in ne 0.
  For i = 1 To OLE1.ObjectVerbsCount - 1
    List1.AddItem OLE1.ObjectVerbs(i)
  Next i

  ' Zapolnimo seznam sprejemljivih oblik vsebine
  For i = 0 To OLE1.ObjectAcceptFormatsCount - 1
    List2.AddItem OLE1.ObjectAcceptFormats(i)
  Next i

  ' Zapolnimo seznam predstavljivih oblik
  ' predmeta v vsebniku
  For i = 0 To OLE1.ObjectGetFormatsCount - 1
    List3.AddItem OLE1.ObjectGetFormats(i)
  Next i

End Sub

Private Sub List1_Click()
  ' Izvedemo izbrani ukaz. Ker smo
  ' prvi ukaz izpustili, prištejemo 1
  OLE1.DoVerb List1.ListIndex + 1
End Sub
```

Metoda DoVerb prejme celoštevilski parameter, ki določa želeno operacijo. Pozitivne vrednosti pripadajo ukazom, značilnim za izbrani razred, ničla predstavlja privzeti ukaz, negativne vrednosti pa so za vse razrede enake in naj bi jih vsi tudi podpirali. Njihov opis najdemo v sprotni pomoči.

Poleg vstavljanja novega predmeta s pogovornim oknom imamo na voljo tudi metodi **CreateLinked** in **CreateEmbedd** za ustvarjanje povezanega oziroma vstavljenega predmeta zgolj s programsko kodo in metodo **PasteSpecialDlg**, ki prikaže pogovorno okno za posebno lepljenje predmeta na odložišču. Vsebovani predmet lahko shranimo v datoteko z metodo **SaveToFile** in ga iz nje preberemo z metodo **ReadFromFile**.

> **NAMIG** Obravnavo avtomatizacije skozi programsko kodo smo prihranili za četrti del knjige, posvečen predmetni naravi visual basica.

Oblikovani vnos

Oblikovano vnosno polje lahko namesto nas skrbi za pravilno *obliko* vnosa najrazličnejših podatkov, kot so datumi, telefonske številke in podobno. Uporabniku prikaže osnovno obliko vnosa, ki ga mora izpolniti. Drugače je oblikovano vnosno polje povsem enakovredno *enovrstični* obliki običajnega vnosnega polja (*TextBox*). Tudi za povezavo s podatkovnim virom uporablja enake lastnosti: DataSource in DataField.

Temeljna lastnost, po kateri se oblikovano vnosno polje razlikuje od običajnega, je **Mask**. Njena vrednost je niz, v katerem vsak znak bodisi opredeljuje dovoljeni vnos bodisi predstavlja običajen znak, ki bo dodatno ponazoril vnos. Pri določanju lastnosti Mask so na voljo naslednji posebni znaki:

Znak	Opis
`A`	Zahteva vnos alfanumeričnega znaka (črke ali cifre).
`a`	Določa neobvezen vnos alfanumeričnega znaka.
`?`	Zahteva vnos črke.
`#`	Zahteva vnos cifre (0–9).
`9`	Določa neobvezen vnos cifre.
`&` ali `C`	Določa neobvezen vnos znaka (znaki s kodo ANSI v območju 32–126 in 128–255, torej tudi presledek).
`.,:/`	Ločila. Ti simboli dodatno ponazorijo vnos.
`>`	Pretvori vse znake, ki sledijo, v velike črke.

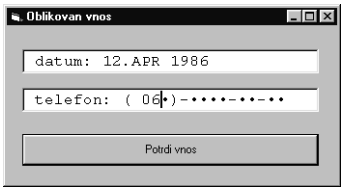
Znak	Opis
<	Pretvori vse znake, ki sledijo, v male črke.
\	Simbol, ki sledi temu znaku, bo izpisan v maski. Tako lahko izpišemo tudi kateregakoli od posebnih simbolov (npr. A, # in 9).

Vsi drugi znaki se neposredno izpišejo in jih uporabnik ne more odstraniti. Zanimivejše lastnosti so še **PromptChar**, ki določa simbol še nevnesenih znakov, viden le, kadar polje prejme vnosno točko. Lastnost **PromptInclude** določa, ali se ta znak odraža tudi v lastnosti Text. Lastnost **ClipMode** pa določa, ali se opisni znaki prikažejo pri prenosu besedila na odložišče.

Če pride do nepravilnega vnosa, se pripeti dogodek **ValidationError**, katerega odzivna procedura prejme do tega trenutka vneseno besedilo in zaporedno mesto neustreznega znaka. Oglejmo si preprost zgled:

```
Private Sub Form_Load()
  With MaskEdBox1
    .Font.Name = "Courier New"
    .Font.Size = 12
    .Mask = "d\atum: ##.>??? ####"
  End With
  With MaskEdBox2
    .Font = MaskEdBox1.Font
    .Mask = "telefon: (9###)-999#-##-##"
    .PromptChar = "•"
  End With
End Sub
```

Slika 11-13: Zgled v rabi



Če želimo izprazniti vsebino oblikovanega vnosnega polja s postavljeno lastnostjo Mask, moramo prazni niz prirediti najprej tej lastnosti, potem pa še lastnosti Text.

Vezana seznama

Tudi vgrajene sezname je mogoče povezati s podatkovnim virom, vendar v tem primeru ponazarjata le vrednost trenutnega zapisa. Vezana seznama, ki ju najdemo v zunanji knjižnici ActiveX, omogočata pregledovanje, popravljanje in vnašanje vrednosti iz celotne zbirke zapisov, kar je zelo uporabno pri najrazličnejših šifrantih. Učinek teh gradnikov je seveda mogoče posnemati tudi z nekaj programske kode in običajnimi gradniki.

Vezana seznama uporabljamo predvsem takrat, ko želimo opisno prikazati opisne vrednosti iz šifranta (*look-up table*), katerega šifra se prepiše v drugo zbirko zapisov. Zato omogočata povezavo z dvema viroma podatkov. Prvi vsebuje vrednosti, ki zapolnijo seznam, drugi pa se popravlja glede na izbiro s seznama.

Za razliko od večine vezanih gradnikov imata vezana seznama kar pet lastnosti, ki opredeljujejo povezavo. To so:

Lastnost	Opis
DataSource	Ime prvega gradnika Data, na katerega je seznam vezan.
DataField	Ime polja v zbirki zapisov, določeni z lastnostjo DataSource *prvega* gradnika Data. To polje določa, kateri element s seznama bo označen. Če uporabnik izbere drug element seznama in se premakne na nov zapis, se bo ustrezno popravila še vrednost tega polja.
RowSource	Ime drugega gradnika Data, ki bo poskrbel za vsebino seznama.
ListField	Ime polja v zbirki zapisov, določeni z lastnostjo DataSource *drugega* gradnika Data, ki bo zapolnilo vrstice v seznamu. To polje *opisuje* nabor vrednosti, ki se lahko priredijo polju, določenemu z lastnostjo DataField.
BoundColumn	Ime polja v zbirki zapisov, določeni z lastnostjo DataSource drugega gradnika Data. To polje mora biti *enakega* tipa kakor polje, določeno z lastnostjo DataField. Ta lastnost *določa* nabor vrednosti, ki se lahko priredijo polju, določenemu z lastnostjo DataField. Če te lastnosti ne postavimo, se privzame enako polje, kakor ga določa lastnost ListField.

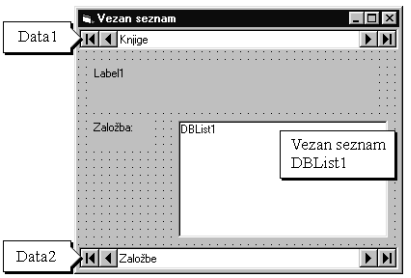
Če želimo, lahko vezana seznama uporabljamo le z enim gradnikom Data. Potem bosta lastnosti DataSource in RowSource ter DataField in ListField postavljeni na isto vrednost. V tem primeru bo gradnik obnavljal tisto zbirko zapisov, ki mu priskrbi vsebino seznama. Če uporabimo odprt seznam, bo nova vsebina polja lahko le ena od že vsebovanih iz vseh preostalih zapisov, pri uporabi kombiniranega seznama pa je mogoče vnesti tudi novo, neodvisno vsebino.

Prav gotovo je za razumevanje delovanja vezanih seznamov potreben zgled. Znova bomo uporabili priloženo zbirko podatkov Biblio.MBD. V njej je poleg tabele knjig tudi tabela (šifrant) založb, ki vsaki založbi dodeljuje enolično številčno oznako. Ta oznaka je shranjena v tabeli knjig v polju z imenom »PubID«. S tem je zbirko podatkov mnogo preprosteje vzdrževati. Če se spremeni naziv založbe, zadošča popravek v šifrantu in novo ime se bo odrazilo v vseh knjigah, ki jih je ta založba izdala. Hkrati prihranimo prostor in olajšamo vnos novih knjig. Kaj pa, če želimo pri kaki knjigi spremeniti založbo? Potem bi morali posodobiti vrednost polja PubID. Vendar je v tem polju šifra. Le kdo si bo zapomnil, kaj pomeni šifra 125 in

kaj šifra 4435? Tu nam pomaga vezani seznam. Njegovo vsebino bomo zapolnili z opisnimi vrednostmi (torej z imeni založb), prepisovali pa bomo dejansko vrednost (šifro založbe).

> **NAMIG** Med tabelo knjig in založb je vzpostavljeno posebno razmerje, ki ga v žargonu imenujemo *relacija*. Uporaba relacijskih zbirk podatkov znatno olajša vzdrževanje podatkov. Njihov podrobnejši opis bomo spoznali pozneje, ko bomo obravnavali jezik SQL.

Pripraviti si moramo uporabniški vmesnik po vzoru s spodnje slike:



Slika 11-14: Vmesnik za zgled rabe vezanega seznama

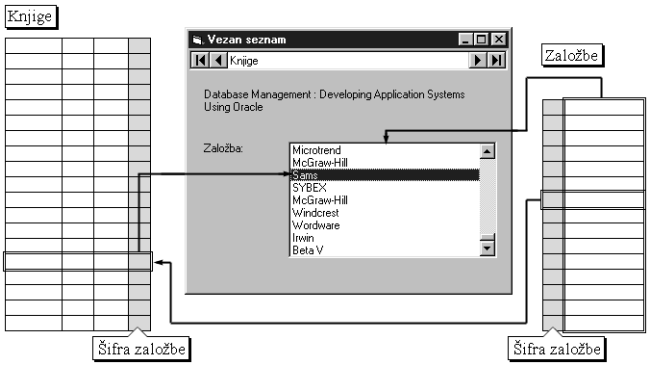
Na njem so gradniki z naslednjimi lastnostmi:

Gradnik	Lastnost	Vrednost
podatki (*Data*)	(Name)	Data1
	DatabaseName	Biblio.MDB
	RecordSource	Titles
	Caption	Knjige
podatki (*Data*)	(Name)	Data2
	DatabaseName	Biblio.MDB
	RecordSource	Publishers
	Visible	False
	Caption	Založbe
oznaka (*Label*)	(Name)	Label1
	DataSource	Data1
	DataField	Title

Gradnik	Lastnost	Vrednost
vezani seznam (*DBList*)	(Name)	DBList1
	DataSource	Data1
	DataField	PubID
	RowSource	Data2
	BoundColumn	PubID
	ListField	Name

In to je vse. Brez ene same vrstice kode smo vzpostavili program za popravljanje povezanih tabel z vpogledom v šifrant. Z zgledom opišimo, kaj se pravzaprav dogaja pri uporabi vezanega seznama. Lastnosti DataSource in DataField določata polje, katerega vrednost se lahko spremeni. Novo vrednost za to polje je mogoče dobiti iz povsem druge zbirke zapisov, vendar mora biti drugo polje *istega* podatkovnega tipa. To polje določata lastnosti RowSource in BoundColumn. Ker pa bi bilo prepisovati nerazumljive šifre za uporabnika zelo neprijazno, lahko seznamu določimo še lastnost ListField. Ta določa *opisno* ime vrednosti, ki se prirejajo.

Slika 11-15: Prikaz delovanja zgleda



Pri uporabi vezanih seznamov nam pomagata še dve zanimivi lastnosti. Lastnost **SelectedItem** vrne oznako (*bookmark*) izbrane vrstice na seznamu. To oznako lahko uporabimo v kodi za doseg celotnega zapisa v zbirki, ki ga določa lastnost RowSource. Na seznamu je namreč vidno le izbrano polje, ki ga določa lastnost ListField. Druga lastnost je **MatchEntry**, ki jo lahko postavimo na vrednost konstante dblExtendedMatching. V tem primeru bo uporabnik lahko preiskoval daljše sezname s tipkovnico. Kadar ima seznam vnosno točko (*focus*), je mogoče s pritiskom na tipko izbrati prvo vrstico, ki se začne z izbranim znakom. Če ustrezno nastavimo še lastnost MatchEntry, bo uporabnik lahko »vnašal« še več znakov, seznam pa bo označeval vrstico, ki se ujema v vseh. Vnos označevanja se »zbriše« ob pritisku na tipko Backspace ali s prenosom vnosne točke na drug gradnik.

Vezani preglednici

Najzmogljivejša vezana gradnika za prikaz podatkov sta preglednici DBGrid in MSFlexGrid. Naenkrat sta sposobna prikazati celotno zbirko zapisov. Značilnosti obeh preglednic sta našteti v spodnji tabeli:

Gradnik DBGrid	Gradnik MSFlexGrid
Uporaben predvsem kot vezani gradnik.	Uporaben predvsem kot nevezani gradnik.
Omogoča povezavo s celotno zbirko zapisov.	Omogoča povezavo s celotno zbirko zapisov.
Omogoča poljubno delo z zapisi.	Omogoča samo pregledovanje zapisov.
Vse celice so v mreži.	Omogoča združevanje celic v večje enote.
Vsak stolpec ima lahko svojo obliko.	Vsaka celica ima lahko svojo obliko.
Omogoča oblikovanje izpisa in izračunane stolpce.	Vključuje metodo za preiskovanje in zamenjevanje besedila.
	Vsaka celica lahko vsebuje besedilo, sliko ali oboje.

Ker nas predvsem zanimajo možnosti za delo z zbirkami podatkov, si bomo podrobneje ogledali preglednico DBGrid. Lastnosti in koda, ki omogoča prikaz vezane vsebine, je povsem enaka za obe preglednici.

> **NAMIG** Zgled uporabe gradnika MSFlexGrid najdemo med priloženimi zgledi visual basica v mapi Samples\PGuide\MSFlexGd\.

Uporaba preglednice DBGrid je lahko zelo preprosta. Na vir podatkov jo povežemo zgolj z nastavitvijo lastnosti DataSource. Ob zagonu projekta bo preglednica vsebovala celotno zbirko zapisov. Vrednost vsakega polja je mogoče trajno popraviti. Manjka le možnost dodajanja in brisanja zapisov, a to dodamo zgolj z nastavitvijo ustreznih lastnosti.

Lastnosti **Row** in **Col**, ki sta dostopni le iz programske kode, opisujeta položaj izbrane celice (vrstico in stolpec). Lahko ju nastavimo ali beremo in s tem ugotovimo, katero celico je izbral uporabnik. Ob spremembi izbrane vrstice preglednica samodejno izbere ustrezen zapis v zbirki. Ta zapis ponazarja lastnost **Bookmark**, ki vsebuje njegovo oznako. Lastnost lahko preberemo in s tem shranimo položaj ali pa nastavimo in s tem sprožimo zapis na prej shranjenem položaju.

Če želimo gradnik natančneje nadzorovati, moramo določiti več predmetov razreda **Column**, zbranih v zbirki Columns, iz katerih je preglednica pravzaprav sestavljena. Lastnosti stolpcev moramo določiti med načrtovanjem s posebnim režimom urejanja, v katerega lahko gradnik preklopimo.

Slika 11-16: Gradnik DBGrid v najpreprostejši rabi z nastavitvijo ene same lastnosti

Lastnosti med načrtovanjem

V najpreprostejši obliki se preglednica sama prilagodi zbirki zapisov, ki ji pripada. Če želimo na njen videz vplivati z nastavitvami med načrtovanjem, moramo najprej vklopiti poseben režim urejanja. Omogočimo ga z izbiro **Edit**... iz priročnega menuja gradnika DBGrid, izrisanega na obrazcu.

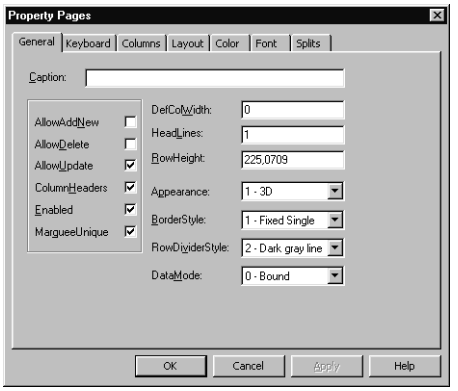
Slika 11-17: Priročni menu gradnika DBGrid

Sedaj je mogoče spreminjati širino posameznega stolpca in višino vrstic, v priročnem menuju pa se prikažeta ukaza **Insert** in **Append** ki omogočata dodajanje stolpcev, ukaz **Delete**, ki izbrani stolpec odstrani, ter ukaza **Cut** in **Copy**, ki omogočata preurejanje položaja. Ko enkrat začnemo preurejati stolpce, moramo ročno določiti tudi njihovo vsebino. Najhitreje gre tako, da iz priročnega menuja izberemo ukaz **Retrieve Fields**. S tem v eni potezi določimo začetni videz stolpcev na podlagi povezane zbirke zapisov.

Druge nastavitve stolpcev omogoča posebna lastnost (Custom), ki prikaže večstransko pogovorno okno (*property pages*) za nastavitev značilnih lastnosti gradnika na enem mestu. Nekatere lastnosti vplivajo na videz celotne tabele, pomembnejše so naštete v spodnji tabeli:

Lastnost	Opis
AllowAddNew	Določa možnost dodajanja novih zapisov. Privzeto neresnično.
AllowDelete	Določa možnost odstranjevanja zapisov. Privzeto neresnično.

Lastnost	Opis
AllowUpdate	Določa možnost popravljanja zapisov. Privzeto resnično.
DefColWidth	Privzeta širina stolpcev.
RowHeight	Višina vrstic.
TabAction	Vloga tipke Tab.



Slika 11-18: Zbrane pomembnejše lastnosti gradnika DBGrid

NAMIG Velikost vrstice in stolpcev podajamo v skali vsebnika, torej obrazca.

Jezička **Columns** in **Layout** skrivata nastavitve za posamezen stolpec. Želeni stolpec najprej izberemo z zaprtega seznama, spodaj pa pozneje določimo njegove nastavitve. Pomembnejše lastnosti za posamezne stolpce so:

Lastnost	Jeziček	Opis
AllowSizing	Format	Določa možnost spreminjanja širine s strani uporabnika.
Locked	Format	Določa možnost popravljanja vsebine v stolpcu.
Alignment	Format	Vodoravna poravnava v stolpcu.
Width	Format	Širina stolpca.
DefaultValue	Columns	Privzeta vrednost polja ob dodajanju zapisa.
NumberFormat	Columns	Oblika izpisa številčne vrednosti (lastnost uporablja enako določilo kakor funkcija Format).
WrapText	Format	Določa izpis besedila, ki je preširoko za stolpec. Če je lastnost nastavljena, se izpis lomi v več vrstic.

Lastnost	Jeziček	Opis
Button	Format	Določa navzočnost gumba v desnem zgornjem vogalu celice. Če kliknemo ta gumb, sprožimo poseben dogodek, ki ga lahko prestrežemo in uporabimo za prikaz dodatnega seznama ali podobnega gradnika.

Gradnik omogoča tudi razcep (*split*) svoje površine v več neodvisnih preglednic. Razcep dosežemo v režimu urejanja z izbiro ukaza **Split** iz priročnega menuja, odstranimo pa ga z ukazom **Remove**. Nastavitve za posamezne razcepe je mogoče določati pod jezičkom **Splits**.

Uporaba iz kode

V programski kodi se z gradnikom DBGrid v glavnem ukvarjamo prek njegove lastnosti Columns, ki je zbirka (*collection*) vseh stolpcev. Poglavitna lastnost te zbirke je **Value**, ki vedno vsebuje vrednost polja v izbrani vrstici. Podobna ji je lastnost Text, ki pa vsebovano vrednost vedno predstavi kot niz znakov. Lastnost lahko beremo in nastavljamo. Morebitno neusklajenost vsebine z vezanim poljem v zbirki podatkov tako kakor pri drugih vezanih gradnikih ponazarja lastnost DataChanged.

Prek zbirke Columns so dostopne tudi vse lastnosti, ki smo jih za posamezen stolpec nastavljali med načrtovanjem. Tako lahko posamezen stolpec skrijemo, zaklenemo, mu spremenimo širino ali obliko izpisa in podobno. Naslednji zgled prikaže vsebino v drugem stolpcu in ga zatem skrije:

```
With DBGrid1.Columns(1)
  MsgBox .Value
  .Visible = False
End With
```

Če spremenimo podatkovni vir, ki priskrbi vsebino preglednice, moramo poklicati metodo **ReBind**, ki obnovi njeno vsebino. Ta metoda spoštuje nastavitev stolpcev med načrtovanjem, kar včasih ni dobro, saj je lahko nov podatkovni vir povsem neprilagojen pripravljenim stolpcem. Pred klicem metode ReBind lahko pokličemo metodo **ClearColumns**, ki bo vzpostavila prvotno stanje. Preglednica se bo potem samodejno prilagodila novi zbirki podatkov, katere vsebino prikazuje. Če želimo po obnovitvi za vsako ceno obdržati lastne stolpce, lahko pred klicem ReBind pokličemo metodo **HoldFields**.

Če smo posameznemu stolpcu nastavili lastnost Button, lahko uporabnik tako, da klikne gumb, ki se prikaže v desnem zgornjem vogalu celice, pokliče odzivno proceduro dogodka ButtonClick. Ta kot parameter prejme zaporedno številko stolpca, tako da je mogoče gumb prirediti tudi več stolpcem.

Na potek popravljanja, obnavljanja, brisanja in vstavljanja zapisov lahko vplivamo z odzivom na posebne dogodke **Before**/**After**, ki se pokličejo tik pred izvršeno

operacijo in tik za njo. Brisanje npr. spremljata dogodka **BeforeDelete** in **AfterDelete**. Vsi dogodki Before prejmejo parameter Cancel, katerega vpliv se odrazi navzven. Če ga nastavimo na resnično vrednost, bomo s tem preklicali izvršitev operacije.

Samodejna obravnava območij ni vključena med sposobnosti gradnika DBGrid. Uporabnik lahko označi območje stolpcev ali vrstic, njegova obdelava pa je prepuščena programski kodi. Denimo, da smo uporabniškemu vmesniku dodali gumb, ki omogoča brisanje vseh izbranih zapisov:

```
Private Sub Command1_Click()
Dim vntBm As Variant

  For Each vntBm In DBGrid1.SelBookmarks
    Data1.Recordset.Bookmark = vntBm
    Data1.Recordset.Delete
  Next
  Data1.Recordset.MoveFirst

End Sub
```

Vidimo, da je oznake izbranih vrstic mogoče razbrati iz zbirke **SelBookmarks**. Če želimo ugotoviti razpon izbranih stolpcev, preberemo lastnosti **SelStartCol** in **SelEndCol**.

> **NAMIG** Več vrstic je mogoče izbrati s klikanjem prvega stolpca stanja (navadno sive barve) ob pritisnjeni tipki Ctrl.

Nevezani način

Gradnik DBGrid lahko uporabimo tudi v *nevezanem* načinu (*unbound mode*). Takrat moramo za vsebino vseh celic skrbeti sami, z odzivom na posebne dogodke. To nam omogoča rabo lastnih zapisov zbirk podatkov ali prilagoditev gradnika poljubnim podatkom. Lahko bi npr. prilagodili gradnik, da bi opisno prikazoval ukaze, ki jih uporablja program Risar, razvit v drugem delu te knjige. Podatke, prikazane v nevezani preglednici, imenujemo *podrejena* podatkovna struktura (*underlying data set*). To je lahko zbirka podatkov, lahko pa gre za poljubno podatkovno strukturo.

Način navezave gradnika na podatke določa lastnost **DataMode**, ki jo lahko nastavimo le med načrtovanjem. V nevezanem načinu se prožijo naslednji dogodki:

Dogodek	Opis
UnboundReadData	Dogodek se sproži, kadar mora gradnik prikazati novo vsebino.

Dogodek	Opis
UnboundGetRelativeBookmark	Dogodek je podoben prejšnjemu, le da se proži za posamezno vrstico, kar lahko vpliva na učinkovitost programa. Sicer ga lahko zanemarimo in obnavljanje vsebine v celoti obdelamo znotraj dogodka UnboundReadData.
UnboundWriteData	Dogodek se sproži, kadar je treba obnoviti vso vrstico v podrejeni podatkovni strukturi.
UnboundAddData	Dogodek se sproži, kadar je v podrejeno podatkovno strukturo treba dodati nov zapis.
UnboundDeleteRow	Dogodek označuje potrebo po brisanju zapisa iz podrejene podatkovne strukture.

Odzivne procedure prejmejo vse potrebne podatke skozi več parametrov, med katerimi je najpomembnejši predmet razreda **RowBuffer**, ki natančno opisuje vse vrstice, ki jih je treba posodobiti. Naša podatkovna struktura mora za vsak zapis določiti enolično oznako, ki se odraža v parametru Bookmark. Ker je ta parameter prilagodljivega tipa, podatkovni tip oznake ni pomemben, pomembna je le njegova enoličnost.

Zgovoren zgled rabe nevezanega gradnika bi zasedel preveč prostora in prekinil tematiko, ki jo obravnavamo. Vsi, ki vas to opravilo zanima, najdete popoln zgled na plošči CD-ROM, priloženi knjigi, v mapi \Primeri\3-Podatki\Nevezana-Preglednica\. Zgled je dovolj splošen, da ga lahko hitro prikrojite lastnim potrebam.

12. POGLAVJE

Predmetni model

Uporaba vezanih gradnikov je privlačna možnost za izdelovanje manj zahtevnih projektov, ki uporabljajo zbirke podatkov. Pravo moč podatkovnega stroja Jet pa je mogoče izkoristiti šele, če uporabimo celotni predmetni model DAO (*Data Access Objects*) in ukaze v standardnem jeziku za delo z zbirkami podatkov, SQL (*Structured Query Language*). Predmetni model DAO opisuje tako zgradbo zbirke podatkov kakor tudi njeno vsebino. Podobno jezik SQL omogoča tako izgradnjo zbirke podatkov kakor njeno poznejšo rabo.

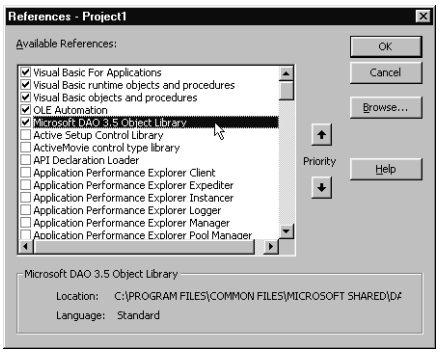
NAMIG Predmetni model DAO omogoča tudi neposreden pristop do sredstev ODBC, ki zaobidejo podatkovni stroj Jet. Več o neposrednem delu s podatkovnimi strežniki bomo izvedeli v posebnem poglavju.

Zgradba

Predmetni model DAO ni neločljivi del visual basica. Preden lahko v kodi uporabimo njegove predmete, moramo omogočiti ustrezno predmetno knjižnico. Če uporabniškemu vmesniku dodamo gradnik Data, se ta knjižnica vklopi samodejno, zato je za zglede v prejšnjem poglavju ni bilo treba ročno vklapljati.

Nove predmetne knjižnice ročno vključujemo v projekt z ukazom **References** iz glavnega menuja **Project**. Prikaže se seznam vseh izpostavljenih predmetnih knjižnic, ki so prijavljene v sistemskem registru, kar prikazuje tudi naslednja slika:

Slika 12-1: Razpoložljive predmetne knjižnice

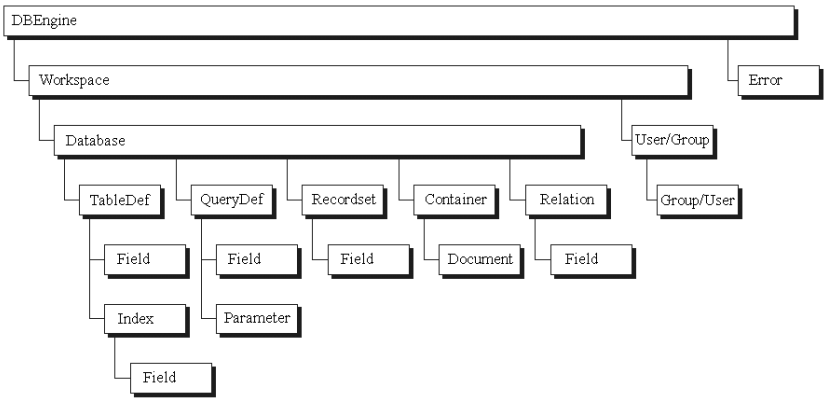


Opazimo lahko, da so tudi osnovni elementi visual basica zbrani v nekaj predmetnih knjižnicah. Za uporabo predmetnega modela DAO na seznamu **Available References** vklopimo knjižnico *Microsoft DAO 3.5 Object Library*. Seznam prikazuje knjižnice v posebnem vrstnem redu. Že izbrane so naštete pri vrhu, preostale pa sledijo po abecednem redu, zato je knjižnica Microsoft DAO na začetku nekoliko globlje na seznamu.

> **NAMIG** Predmetna knjižnica DAO, priložena visual basicu 5, je 32-bitna različica z oznako 3.5. Združljiva je s 16/32-bitno različico 3.0, ki je spremljala visual basic 4. Za združljivost s še starejšimi predmetnimi modeli DAO je na voljo posebna knjižnica *Microsoft DAO 2.5/3.5 Compatibility Library*.

Izraz predmetni model (*object model*) preprosto pomeni le to, da lahko neki predmet vsebuje nove predmete, ti pa lahko spet vsebujejo nove itn. Predmetni model DAO je precej razvejen. Obsega 17 razredov, ki so vsi, razen osnovnega, **DBEngine**, vsebovani v zbirkah (*collections*). Shematski prikaz predmetnega modela je razviden z naslednje slike:

Slika 12-2: Predmetni model DAO



Ime zbirke je vedno angleška oblika množine za razred predmetov, ki jih vsebuje. Predmeti razreda **Workspace** so tako zbrani v zbirki Workspace**s**, predmeti razreda **Index** pa v zbirki Index**es**[49].

Zaradi preprostejše rabe in tudi zaradi združljivosti s starejšo kodo je predmet DBEngine vedno na voljo. Na voljo je samo v enem, vnaprej pripravljenem izvodu. V njem lahko izdelamo več delovnih prostorov (*workspaces*). Privzeti delovni prostor, prvi element zbirke – `Workspaces(0)` – je prav tako vedno na voljo, saj se vzpostavi samodejno, po potrebi.

[49] Pravzaprav je v angleščini običajnejša množina besede Index Indices. Zaradi enotnega obravnavanja so ne glede na slovnične podrobnosti vse zbirke poimenovane z zaključnim *-s* oziroma *-es*.

Opis zbirke podatkov

Najprej bomo spoznali elemente DAO, ki se ukvarjajo z *definicijo podatkov*. Definicija podatkov (*data definition*) je le drug izraz za zgradbo zbirke podatkov. Osnovne elemente zbirke podatkov smo že spoznali v uvodnem delu tega poglavja. Tu si bomo ogledali, kako lahko uporabimo predmete DAO za izgradnjo ali spremembo zgradbe zbirke podatkov.

Zbirke podatkov je mogoče *organizirati* na več načinov. Poznamo zaporedne, hierarhične, predmetne, hibridne in relacijske načine organizacije. Organizacija zbirke ne določa samo načina, kako so podatki shranjeni, temveč predstavlja logični model, ki ga uporablja programer pri njihovi rabi. Danes lahko brez dvoma trdimo, da je relacijski model postal splošno veljaven standard. Njegov uspeh temelji predvsem na moči in razprostranjenosti jezika SQL, ki omogoča učinkovito modeliranje in izkoriščanje podatkov. Tudi podatkovni stroj Jet, ki si ga visual basic deli z orodjem Microsoft Access in vedno več sistemskimi storitvami operacijskega sistema Okna, uporablja relacijski podatkovni model in s tem podpira jezik SQL.

Načrtovanje relacijske zbirke

Načrtovanje relacijske zbirke podatkov zahteva porazdelitev podatkov v relacijsko povezane tabele. Dobro načrtovana zbirka ima tabele, ki ne vsebujejo podvojenih podatkov. Načrtovanju take zbirke pravimo *normalizacija*. Normalizacija (*normalization*) je lahko zelo zapleten proces, odvisno pač od narave problema. Učinkovito izdelana zbirka omogoča hitro preiskovanje in enostavno vzdrževanje.

Oglejmo si preprost zgled. Denimo, da bi radi v zbirko podatkov shranili podatke o knjigah (taka zbirka je datoteka Biblio.MDB, ki jo uporabljamo v večini zgledov). Osnovni podatki o knjigah, zapisani v zaporedni zbirki, bi bili videti takole:

Avtor	Naslov	Založba
M. Cof	Predmetno programiranje	**ABC Knjige**
P. Tabelnik	Knjiga o SQL	**ABC Knjige**
M. Cof, P. Tabelnik	Odprimo Okna	Libris

Pri samo treh knjigah se je dvakrat ponovilo ime avtorja in založbe. Verjetno bi bilo bolj učinkovito, če bi založbe zapisali v posebno tabelo in vzpostavili *relacijo*. Denimo takole:

Tabela knjig

Avtor	Naslov	Založba
M. Cof	Predmetno programiranje	1
P. Tabelnik	Knjiga o SQL	1
M. Cof, P. Tabelnik	Odprimo Okna	2

Tabela Založb

Založba	Opis
1	ABC Knjige
2	Libris

Relacija

Tabela knjig ➔ Tabela založb, eden z enim po polju Založba.

Podobno bi lahko opravili tudi z avtorji, le da ima lahko ena knjiga tudi več avtorjev. Zato moramo poseči po vmesni tabeli, ki preslika vsakega avtorja v vse knjige, ki jih je napisal. Kako je to dejansko narejeno, si bomo ogledali pri analizi zbirke Biblio.MDB.

Pri uporabi relacijskih zbirk podatkov naletimo na pojem *ključ*. Ključ (*key*) je polje ali kombinacija polj v tabeli, ki so navadno indeksirana zaradi hitrega preiskovanja. Ključ je lahko enoličen ali ne, odvisno od primera. V zgornjem zgledu bi moral biti ključ po polju Založba za tabelo založnikov vsekakor enoličen, pri tabeli knjig pa ne, saj se enaka založba lahko ponovi pri več knjigah. Samo enolično določen ključ lahko določimo kot *primarni*. Primarni ključ (*primary key*) določa natančno en zapis v tabeli.

Relacije (*relation*) omogočajo medsebojno povezovanje tabel po ključih. V zgornjem zgledu smo relacijsko povezali tabeli knjig in založb, da smo dobili prvotno obliko zapisa. Tabeli smo povezali po ključu »Založba«, ki je za tabelo knjig *tuji* ključ (*foreign key*), saj predstavlja primarni ključ druge tabele. Vsaki knjigi pripada ena založba, zato je vzpostavljena relacija vrste »eden z enim« (*one-to-one*). Kaj pa, če bi želeli izpisati vse knjige, ki jih je izdala posamezna založba? Tedaj bi lahko vzpostavili relacijo »eden z mnogimi« (*one-to-many*) v nasprotni smeri. V določenih primerih imamo lahko tudi relacijo »mnogi z mnogimi« (*many-to-many*). Zgled take relacije bi bila povezava avtorjev s knjigami. Tovrstne relacije, kakor sem že omenil, rešimo z dvema relacijama »eden z mnogimi« s tretjo tabelo, ki obsega povezane tuje ključe obeh relacij.

Orodje Visual Data Manager

Visual basicu je priloženo priročno orodje (razvito kar v visual basicu, tako kakor Application Setup Wizard, Text API Viewer in še nekateri pripomočki) Visual Data Manager, ki omogoča izdelovanje in pregledovanje zbirk podatkov v različnih oblikah zapisa, ki jih podpira podatkovni stroj Jet.

Orodje je mogoče zagnati iz razvojnega okolja. Iz glavnega menuja **Add-Ins** izberemo ukaz **Visual Data Manager**... Mi ga bomo uporabili za pregledovanje strukture zbirke podatkov Biblio.MDB, ki jo odpremo z menujskim ukazom **File ➪ Open Database ➪ Microsoft Access**... Našli jo bomo v isti mapi kakor visual basic.

Slika 12-3: Orodje Visual Data Manager

V oknu lahko pregledujemo vse elemente zbirke podatkov. Vidimo, da zbirka obsega štiri tabele. Iz priročnega menuja vsake tabele lahko izberemo ukaz **Design**..., ki nam prikaže njeno notranjo zgradbo. Preglejmo najprej tabelo Titles (enakovredna naši tabeli knjig iz prvega zgleda):

Slika 12-4: Zgradba tabele knjig

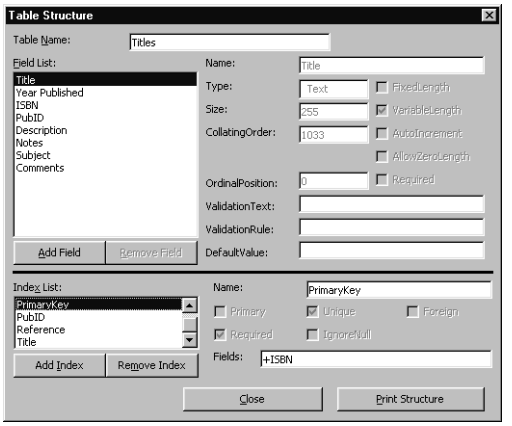


Tabela ima primarni ključ, to je polje ISBN. Oznaka knjige ISBN je enolično določena za ves svet, zato je povsem naraven primarni ključ knjig. Tabela ima še več ključev, ki so uporabni v relacijah. Oglejmo si še tabelo Title Author. Že njeno ime nakazuje vlogo, ki jo ima. Ta tabela pomaga vzpostaviti relacije »mnogi z mnogimi«, zato obsega le dve polji, ki sta tuja ključa za relacijo s tabelo avtorjev in tabelo naslovov oz. knjig.

Z orodjem Visual Data Manager lahko natančno pregledujemo še druge tabele, hkrati pa nam omogoča učinkovito demonstracijo različnih zbirk podatkov in vezanih gradnikov. V vrstici gumbov lahko izbiramo tri vrste predstavitve podatkov in tri različne prikaze vsebine. Z dvakratnim klikom imena tabele ali z izbiro ukaza **Open** iz priročnega menuja se prikaže okno za pregledovanje in popravljanje vsebine zapisov. Orodje ponuja še cel kup pripomočkov, od orodij za vzdrževanje zbirke do vizualnega sestavljanja poizvedb in vnosnih obrazcev. Skratka, izplača se ga preučiti in uporabljati.

V nadaljevanju te knjige se bomo osredotočili na izdelovanje in vzdrževanje zbirke podatkov izključno iz programske kode s predmeti DAO, pozneje pa tudi s stavki SQL.

Izgradnja

Izgradnja nove tabele je pravzaprav vzpostavitev predmeta razreda **Database**. Tabele zbirke določajo predmeti razreda **TableDef**, zbrani v zbirki TableDefs. Vsako tabelo natančneje opredeljujeta zbirki polj in indeksov. Vsako zbirko zapolnimo z enim ali več klici metode **Append**, ki v zbirko doda nov predmet. Preden pa predmet lahko dodamo v zbirko, ga moramo pravilno vzpostaviti.

Postopoma si bomo ogledali zgled, ki samo s predmeti DAO zgradi novo zbirko podatkov. Delo začnemo z izbiro delovnega prostora (*workspace*). Uporabili bomo privzeti delovni prostor, ki se samodejno pripravi, kadar je treba[50]:

```
' Delovni prostor
Dim ws As Workspace
  Set ws = DBEngine.Workspaces(0)
```

Sledi stvaritev nove, zaenkrat še prazne zbirke podatkov:

```
' Zbirka podatkov
Const strDbDatoteka As String = "Zbirka.MDB"
Dim db As Database
  If Dir(strDbDatoteka) <> "" Then Kill (strDbDatoteka)
  Set db = ws.CreateDatabase(strDbDatoteka, dbLangSlovenian)
```

Novo, domorodno zbirko podatkov ustvarimo s klicem metode **CreateDatabase**. Njena popolna oblika je:

[50] To je ob prvem sklicevanju nanj, bodisi neposrednem bodisi posrednem z uporabo katere izmed njegovih metod brez naslavljanja delovnega prostora.

CreateDatabase(*ime*, *jezik*[, *izbire*])

Ime zbirke podatkov in jezik sta obvezna parametra. *Ime* določa ime datoteke MDB, ki je lahko podana skupaj s potjo, *jezik* pa določa pravila urejanja podatkov. Pri izbiranju jezika imamo na voljo tudi slovenščino. Nastavitev izbir ni obvezna, z njimi pa lahko določimo, da se baza ustvari v kateri od starejših oblik zapisa MDB in njeno morebitno *kriptiranje*. S kriptiranjem (*encryption*) dosežemo, da s pregledovanjem »surove« datoteke v zunanjih orodjih (npr. v urejevalniku besedila) ne bo mogoče razbrati v njej shranjenih podatkov. Popoln opis metode najdemo v sprotni pomoči.

Metoda CreateDatabase sproži izvajalno napako, če zbirka podatkov z istim imenom že obstaja. Zato smo pred njenim klicem preverili morebitni obstoj datoteke, ki jo v tem primeru odstranimo.

> **NAMIG** Če ime datoteke za shranjevanje domorodne zbirke podatkov ne vsebuje poti, metoda CreateDatabase uporabi dejavno pot. Izvajalno okolje visual basica poskrbi, da je dejavna pot med delom vedno njegova domača mapa. V tej mapi najdemo tudi zbirki podatkov Biblio.MDB in Nwind.MDB. Ta predpostavka pa ne velja več, če program prevedemo v samostojno izvršljivo datoteko. V tem primeru moramo programsko natančno določiti pot do datoteke z zbirko podatkov. Najpogosteje nam pri tem pomaga lastnost Path vgrajenega predmeta App.

Zbirki bomo dodali tabelo:

```
' Tabele
Dim td As TableDef

  ' - Založbe
  Set td = db.CreateTableDef("Zalozbe")
```

Tako ustvarjena tabela še ni del zbirke podatkov. Najprej jo moramo popolnoma določiti in jo zatem eksplicitno dodati v zbirko. Tabeli najprej določimo polja zapisa:

```
' Polja
Dim fld As Field

    ' - Oznaka založbe
    Set fld = td.CreateField("Zalozba", dbLong)
    fld.Attributes = dbAutoIncrField
    td.Fields.Append fld

    ' - Naziv založbe
    Set fld = td.CreateField("Opis", dbText, 100)
    td.Fields.Append fld
```

Tudi pri polju velja, da ga najprej pripravimo v novem izvodu predmeta razreda **Field**, ki mu določimo vse želene lastnosti in ga zatem eksplicitno dodamo v zbirko

polj Fields z metodo Append. Konstante, ki opisujejo razpoložljive tipe polja, so naštete v sprotni pomoči. V domorodnih zbirkah podatkov je na voljo vsak osnovni podatkovni tip visual basica ter še nekateri posebni tipi, ki jih bomo spoznali pozneje.

Pri polju »Zalozba« vidimo, kako ustvarimo polje, primerno za primarni ključ. Polju tipa Long lahko z lastnostjo **Attributes** določimo, da se bo njegova vrednost samodejno povečevala in bo tako enolično določena za vsak zapis. Konstante, ki jih lahko kombiniramo pri lastnosti Attributes za predmete razreda Field, so zbrane v spodnji tabeli:

Konstanta atributa polja	Opis
dbAutoIncrField	Vrednost polja je nespremenljivo, enolično dolgo število tipa Long, ki se samodejno povečuje za nove zapise. Na voljo je le za domorodne zbirke podatkov.
dbDescending	Polje se ureja v padajočem zaporedju (Ž–A oziroma 100–0). Nastavitev uporabimo pri poljih, ki jih dodamo v indeks.
dbUpdatableField	Vrednost polja se lahko spremeni.
dbVariableField	Dolžina polja se prilagaja vsebini (samo polja vrste Text).
dbFixedField	Dolžina polja je nespremenljiva (privzeto pri številskih poljih).
dbHyperlinkField	Polje vsebuje hiperpovezave (samo polja vrste Memo).
dbSystemField	Polja ni mogoče odstraniti, ker vsebuje informacijo o *replikah*.

Sedaj določimo še indeks, ki ureja podatke po primarnen ključu:

```
' Indeksi
Dim idx As Index

    ' - Indeks za primarni ključ založba
    Set idx = td.CreateIndex("Zalozba")
    idx.Primary = True
    idx.Unique = True
    Set fld = idx.CreateField("Zalozba")
    idx.Fields.Append fld
    td.Indexes.Append idx
```

Določanje indeksa zahteva kar nekaj nastavitev. Najprej vzpostavitev predmeta razreda **Index**, zatem pa dodajanje enega ali več predmetov razreda Field, ki določajo polja, vsebovana v indeksu. Vsakega posebej moramo dodati v zbirko z metodo Append. Ta polja morajo biti izbrana izmed prej definiranih polj tabele. Ne določamo posebej njihovega tipa in velikosti, temveč le njihovo ime in po želji še atribut za urejanje v padajočem zaporedju. Na koncu tako določen indeks dodamo še v zbirko Indexes same tabele. Popolnoma določeno tabelo dodamo v zbirko TableDefs in s tem v zbirko podatkov:

```
db.TableDefs.Append td
```

Pripravimo še tabelo knjig:

```
' - Knjige
Set td = db.CreateTableDef("Knjige")

  ' Polja
  ' - ISBN
  Set fld = td.CreateField("ISBN", dbText, 10)
  td.Fields.Append fld

  ' - Naslov
  Set fld = td.CreateField("Naslov", dbText, 100)
  td.Fields.Append fld

  ' - Avtor
  Set fld = td.CreateField("Avtor", dbText, 100)
  td.Fields.Append fld

  ' - Oznaka založbe
  Set fld = td.CreateField("Zalozba", dbLong)
  td.Fields.Append fld

  ' Indeksi
  ' - Indeks za primarni ključ
  Set idx = td.CreateIndex("ISBN")
  idx.Primary = True
  idx.Unique = True
  Set fld = idx.CreateField("ISBN")
  idx.Fields.Append fld
  td.Indexes.Append idx

  ' - Indeks tujega ključa
  Set idx = td.CreateIndex("Zalozba")
  idx.Primary = False
  idx.Unique = False
  Set fld = idx.CreateField("Zalozba")
  idx.Fields.Append fld
  td.Indexes.Append idx

db.TableDefs.Append td
```

V zbirki podatkov smo določili dve tabeli, ki ju lahko med seboj povežemo, da dobimo izčrpnejšo informacijo. To nam omogoča povezava med primarnim ključem tabele založb in tujim ključem tabele knjig. Da pa bo povezava uporabna, je zelo pomembno vzdrževati referenčne *celovitosti* podatkov pri dodajanju, spreminjanju ali odstranjevanju zapisov. Celovitost podatkov (*referential integrity*) preprosto pomeni, da mora imeti vsaka vrednost tujega ključa tudi par v primarnem ključu povezane tabele. V našem primeru to pomeni, da mora vsaki oznaki založbe v tabeli knjig pripadati tudi ustrezen opis založbe v tabeli založb. Če bi v tabeli založb lahko odstranili opis, na katerega se sklicuje kaka knjiga, bi s tem porušili celovitost podatkov.

Relacije

V podatkovnem modelu DAO je za vsiljevanje celovitosti podatkov predviden predmet razreda **Relation**. Z njim lahko preprečimo rušenje celovitosti podatkov. Zgoraj omenjeni zgled, odstranjevanje založbe, na katero se sklicuje knjiga, bi v tem primeru povzročil izvajalno napako. Oglejmo si vzpostavljanje relacije:

```
' Relacije
Dim rel As Relation

  Set rel = db.CreateRelation("ZalozbeKnjige")
  rel.Table = "Zalozbe"
  rel.ForeignTable = "Knjige"
  Set fld = rel.CreateField("Zalozba")
  fld.ForeignName = "Zalozba"
  rel.Fields.Append fld
  db.Relations.Append rel
```

Pri relaciji določimo tabelo s primarnim ključem in povezano tabelo, katere tuji ključ se nanaša nanj. Podobno kakor pri indeksu moramo tudi pri relaciji ustvariti predmet razreda Field, ki določa povezana ključa. Z metodama Append najprej dodamo polje v relacijo, zatem pa še določeno relacijo v zbirko podatkov. Tudi predmet razreda Relation pozna lastnost Attributes, ki je zgoraj sicer nismo uporabili. Na voljo je nekaj konstant, ki znatno vplivajo na vedenje vzpostavljene relacije. To so:

Konstanta atributa relacije	Opis
dbRelationUnique	Določa relacijo »eden z enim«.
dbRelationDontEnforce	Relacija ne vsiljuje celovitosti podatkov.
dbRelationUpdateCascade	Relacija vsili *kaskadno* dopolnjevanje. Če se spremeni vrednost primarnega ključa, se v povezani tabeli spremenijo tudi vrednosti tujega ključa vseh zapisov, ki mu ustrezajo.
dbRelationDeleteCascade	Relacija vsili kaskadno odstranjevanje. Če odstranimo primarni ključ, se v povezani tabeli odstranijo tudi vsi zapisi, ki vsebujejo odgovarjajoč tuji ključ.

Na koncu moramo zbirko podatkov pravilno zapreti:

```
db.Close
```

NAMIG Videli smo, da je izgradnja zbirke podatkov v programski kodi precej naporno opravilo, ki pa ponuja popoln nadzor nad vsemi njenimi elementi. Pozneje bomo spoznali še lažji način ustvarjanja zbirke podatkov s stavkom v SQL. Ko bomo obravnavali jezik SQL,

bomo spoznali tudi možnost dodajanja izdelanih, parametriziranih poizvedb v strukturo zbirke podatkov. Določajo jih predmeti zbirke QueryDefs.

Celotno programsko kodo zgoraj opisanega zgleda in drugih zgledov v tem poglavju najdete na priloženi plošči CD-ROM v mapi \Primeri\3-Podatki\1-Predmetni model\.

Celovitost zapisa

Nekaterih elementov določanja strukture zbirke podatkov, kot so pravila vnosa (*validation rules*), v zgoraj obravnavanem zgledu nismo omenili. Z nekaj lastnostmi predmetov TableDef in Field lahko vsako polje opremimo z dodatnimi določili, ki omejujejo njegov nabor vrednosti. Tako lahko npr. preprečimo, da bi kako polje ob vnosu novega zapisa ostalo prazno ali pa z lastnim območjem omejimo sprejemljive vrednosti. Lastnosti predmeta Field, ki nam to omogočajo, so opisane v spodnji tabeli:

Lastnost	Opis
Required	Če to lastnost postavimo, mora biti vsebina polja vedno določena.
AllowZeroLength	Lastnost določa, ali dopuščamo prazen niz kot dovoljeno vsebino polja vrste dbText ali dbMemo.
DefaultValue	Določa privzeto vsebino polja ob dodajanju novega zapisa.
ValidateOnSet	Z nastavitvijo te lastnosti pri vnosu novih vrednosti vsilimo pravilo, ki ga določa lastnost ValidateRule.
ValidationRule	Lastnost določa pravilo vnosa novih vrednosti v polje. Pravilo zapišemo v obliki niza, ki vsebuje logični izraz v jeziku SQL (izraz, ki sledi določilu WHERE). Jezik SQL bomo podrobneje spoznali v naslednjem poglavju.
ValidationText	Niz, shranjen v tej lastnosti, pojasnjuje pravilo vnašanja. Če uporabnik ob vklopljenem preverjanju vnese novo, a neustrezno vrednost, se sproži izvajalna napaka, katere opis je enak tej lastnosti. Če lastnosti ne določimo, je pojasnilo enako pravilu, določenemu z lastnostjo ValidationRule.

Lastnost ValidationRule lahko predpišemo vsakemu polju in s tem omejimo vrednosti, ki jih polje sprejme. Naslednji zgled omeji vrednosti, ki jih sprejme polje, predstavljeno s predmetno spremenljivko fld:

```
fld.ValidationRule = ">=0 AND <256"
```

Izraze za preverjanje vnosa lahko zapišemo samo v tej posebni obliki, kjer bi na levi strani primerjalnega izraza morala biti vrednost polja, ki pa je ne zapišemo. Če je pravilo vnosa odvisno tudi od drugih polj ali moramo preverjanje vsebine nekega polja uporabiti v bolj zapletenem izrazu, lahko uporabimo lastnosti ValidationRule in ValidationText za celotno tabelo. Denimo, da je ta predstavljena s predmetom td:

```
td.ValidationRule = "[Datum oddaje] >= [Datum sprejema]"
```

Oglejmo si še zgled, ki dopolni definicijo zbirke podatkov knjig. Posodobili bomo definicijo polja ISBN, ki bo zahtevalo vnos besedila dolžine natančno 10 znakov in določalo njegovo privzeto vrednost:

```
Set fld = td.CreateField("ISBN", dbText, 10)
With fld
  .Required = True
  .AllowZeroLength = False
  .DefaultValue = "0000000000"
End With
td.Fields.Append fld

' ...

td.ValidationRule = "Len([ISBN]) = 10"
td.ValidationText = "Polje ISBN mora biti dolgo natančno 10 znakov!"

db.TableDefs.Append td
```

Pri vnosu nove, neustrezne vrednosti se bo sprožila izvajalna napaka, ki jo lahko prestrežemo in se nanjo ustrezno odzovemo. Če želimo neustrezno popraviti vrednost obstoječega zapisa, pa se prikaže opozorilo, kot ga določa besedilo lastnosti ValidationText, tako kakor prikazuje spodnja slika.

Slika 12-5: Opozorilo o nepravilnem vnosu

Na celovitost zapisa posredno vplivajo tudi vzpostavljeni indeksi. Pri polju ISBN npr. zaradi indeksa ni mogoč vnos podvojenih vrednosti.

Urejanje

Pri izgradnji zbirke podatkov smo videli, da je ta logično predstavljena s kopico predmetov, združenih v zbirke (*collections*). Njihovo shemo smo si imeli priložnost ogledati pri prvi obravnavi zgradbe predmetnega modela zgoraj. Uporaba in prilagajanje teh zbirk nam omogoča spreminjanje strukture same zbirke podatkov. V nadaljevanju si bomo ogledali metode in lastnosti za spreminjanje strukture, na koncu pa bomo spoznali tudi priklop zunanjih tabel v zbirko podatkov. Kot smo že spoznali, omogoča podatkovni stroj Jet uporabo najrazličnejših oblik zapisovanja podatkov.

Preurejanje zbirke

Tudi preurejanje zbirke bomo ponazorili z zaokroženim zgledom, ki posodobi prej vzpostavljeno zbirko knjig in založb. Seveda sam pojem »preurejanje« predpostavlja, da bomo uporabili že prej ustvarjeno in odprto zbirko podatkov. Zato si najprej oglejmo, kako obstoječo zbirko odpremo. Tu nam pomaga metoda delovnega prostora, imenovana **OpenDatabase**. Simbolični zapis njene rabe je:

```
Set zbirka = OpenDatabase(ime [, opcije] [, samoBranje] [, povezava])
```

Parametri metode skoraj v popolnosti odgovarjajo nekaterim lastnostim gradnika Data. Ime ustreza lastnosti DatabaseName, samoBranje lastnosti ReadOnly in povezava lastnosti Connect. Te smo že opisali, ko smo obravnavali gradnik, še izčrpnejšo informacijo pa najdemo v sprotni pomoči. Posebnost so le izbire, značilne za odpiranje zbirke iz vira ODBC.

Začnimo naš zgled z odpiranjem domorodne zbirke v privzetem načinu:

```
Dim db As Database
Dim td As TableDef
Dim fld As Field

  Set db = OpenDatabase("Zbirka.MDB")
  If db.Updatable Then
    Set td = db.TableDefs("Zalozbe")
    If td.Updatable Then
```

Zaradi združljivosti s starejšo kodo in krajšega zapisa je mogoče metode, ki se sklicujejo na privzeti delovni prostor, zapisati kar neposredno (podobno kakor lahko znotraj kode obrazca neposredno uporabljamo njegove lastnosti in metode). To smo izkoristili v zgornji vrstici kode, ki odpre zbirko podatkov.

Pred preurejanjem se je dobro prepričati, ali je za izbrani predmet sploh mogoče. Zbirka je morda odprta v samo bralnem načinu in ne omogoča spreminjanja ali pa uporabljamo obliko zapisa, ki popravkov sploh ne omogoča. Predmeta razredov Database in TableDef poznata lastnost **Updatable**, ki določa možnost popravljanja. Oglejmo si dodajanje polj v obstoječo tabelo:

```
      Set fld = td.CreateField("Naslov", dbText, 50)
      td.Fields.Append fld

      Set fld = td.CreateField("Telefon", dbText, 20)
      td.Fields.Append fld

    End If
```

Preurejanje zbirke podatkov je v marsičem podobno njenemu vzpostavljanju. Uporabljamo enake predmete in preurejamo enake zbirke, le da poleg metode Append, ki doda nov predmet v zbirko, uporabljamo še metodi **Delete**, ki predmet iz zbirke odstrani, in **Refresh**, ki obnovi stanje nadrejenega predmeta, ki mu posodobljena zbirka pripada.

Posebnih metod za preoblikovanje polj ni. Če bi npr. želeli spremeniti tip polja »Telefon« iz besedila v številčno vrednost, bi morali ustvariti novo tabelo z ustreznimi polji, vanjo prepisati obstoječe podatke in staro tabelo odstraniti.

Pri odstranjevanju elementov moramo biti pazljivejši kakor pri dodajanju. Če želimo odstraniti polje, to ne sme biti vsebovano v nobenem indeksu ali relaciji. Tudi tabela ali indeks, ki ga ne potrebujemo več, ne sme biti vsebovan v katerem od obstoječih predmetov Relation. Odstranjevanje poljubnega predmeta iz zbirke dosežemo s klicem metode Delete:

```
    db.Relations.Delete "ZalozbeKnjige"
    db.TableDefs("Zalozbe").Indexes.Delete "Zalozba"
    db.TableDefs("Zalozbe").Fields.Delete "Zalozba"
    db.TableDefs.Delete "Zalozbe"

  End If
```

Pravkar opravljene posodobitve strukture se odrazijo v nadaljevanju dela. Previdni moramo biti le v večuporabniških okoljih. Trenutno stanje zbirke osvežimo s klicem metode Refresh. Prepogosto klicanje te metode lahko zmanjša hitrost izvajanja, zato jo uporabimo le, kadar je nujno treba.

Na koncu moramo odprto zbirko zapreti:

```
db.Close
```

Zapiranje dosežemo tudi s prirejanjem predmetne spremenljivke neveljavni vrednosti. Ko smo obravnavali predmete, smo že omenili, da ti izginejo, ko se nanje ne sklicuje nobena spremenljivka več:

```
Set db = Nothing
```

> **NAMIG** Če odpremo katero od ustvarjenih domorodnih zbirk podatkov in se z zanko For Each sprehodimo prek vseh obstoječih tabel (predmetov razreda TableDefs), ugotovimo, da vsaka zbirka vsebuje tudi nekaj sistemskih tabel, katerih ime se začenja z »MSys«. Te tabele pomagajo podatkovnemu stroju voditi evidenco o uporabnikih, skupinah, dostopnih pravicah in drugih shranjenih elementih.

Priklop zunanjih tabel

Podatkovni stroj Jet prepozna več različnih oblik zapisa zbirk podatkov. Z njim lahko v vsaki odprti zbirki podatkov uporabimo *priključene* tabele (*linked tables*, včasih poimenovane tudi *attached tables*). Te tabele izvirajo iz druge zbirke podatkov, ki je lahko zapisana v poljubni podprti obliki, v programski kodi pa jih uporabljamo tako, kakor bi bile sestavni del zbirke. Pri priključenih tabelah so določene omejitve, iz njih npr. ne moremo izdelati zbirke zapisov vrste Table, določene oblike

zapisa ne podpirajo posameznih podatkovnih tipov itn. Z nekaj več previdnosti pri kodiranju lahko vse te razlike zaznamo in upoštevamo.

Priklop zunanje tabele je zelo podoben ustvarjanju nove, le da moramo določiti dve dodatni lastnosti. Priključeni tabeli ne moremo dodajati novih polj in indeksov. Oglejmo si zgled, ki priključi tabelo iz zunanje domorodne zbirke podatkov. Uporabili bomo kar visual basicu priloženo domorodno zbirko Biblio.MDB in njeno tabelo avtorjev priključili v prej ustvarjeno lastno zbirko:

```
Dim db As Database
Dim td As TableDef

  Set db = OpenDatabase("Zbirka.MDB")
  Set td = db.CreateTableDef("Avtorji")
  td.SourceTableName = "Authors"
  td.Connect = ";DATABASE=Biblio.MDB;"
  db.TableDefs.Append td

db.Close
```

Določiti moramo lastnosti **SourceTableName** in **Connect**. Prva lastnost določa ime tabele v zunanji zbirki, druga pa opisuje njeno obliko in ime. Opis razpoložljivih oblik smo spoznali, ko smo obravnavali lastnosti Connect gradnika Data. Poleg imena zbirke moramo pri nekaterih posebnih oblikah zapisa podati še kak parameter (npr. dostopno geslo kot parameter PWD).

Tako priključena tabela ostane v definiciji zbirke podatkov tudi po tem, ko jo zapremo. Če je ne potrebujemo več, jo iz zbirke TableDefs odstranimo z metodo Delete. Torej se vede popolnoma enako kakor druge tabele zbirke.

> **NAMIG** Pri uporabi virov ODBC prek podatkovnega stroja Jet je vedno učinkoviteje uporabiti priključene tabele, kakor pa odpirati vir ODBC neposredno.

Vzdrževanje zbirke

Domorodna zbirka podatkov podatkovnega stroja Jet je zapisana v obliki datoteke .MDB, ki shranjuje vse njene elemente. Za večjo varnost podatkov je priporočljivo izdelovati rezervne kopije datoteke (*backup*). Zaradi čim večje hitrosti dela podatkovni stroj ne skrbi za optimalno velikost datoteke, temveč označuje njene neuporabljene dele[51] in na konec datoteke dodaja novo vsebino. Zato se velikost datoteke pri intenzivni rabi lahko precej napihne, hkrati pa se poveča notranja *razdrobljenost* podatkov (*internal fragmentation*) ter s tem zmanjša učinkovitost njihove rabe. Če pri neposrednem delu z zbirko podatkov pride do nepričakovanega

[51] Najmanjši del datoteke domorodne zbirke podatkov je *stran* (*page*); velik je 2048 zlogov.

razpada sistema, bodisi zaradi programske napake bodisi zaradi zunanjega vzroka (npr. izpada električne energije), se lahko del podatkov pokvari. Pokvarjene podatke (*corrupted data*) je večinoma mogoče odkriti, popraviti ali odstraniti iz zbirke. Vsa našteta opravila sodijo k vzdrževanju zbirke podatkov, ki si ga bomo natančneje ogledali.

Zgoščevanje

Zgoščevanje zbirke podatkov optimalizira njeno velikost in odpravi notranjo razdrobljenost. S tem se poveča učinkovitost dela in zmanjša zasedeni prostor na disku. Zgoščujemo lahko le zaprte zbirke podatkov. Zgoščevanja ni mogoče vključiti v *transakcijo* in ga pozneje preklicati. Zgoščevanje opravi metoda predmeta DBEngine z imenom **CompactDatabase**. Njena popolna oblika je:

```
[DBEngine].CompactDatabase izvor, cilj [, jezik] [, opcije]
```

Podobno kakor druge metode tega predmeta jo lahko pokličemo tudi neposredno. Nujno ji moramo posredovati ime obstoječe, izvorne ter ciljne, nove datoteke z zbirko podatkov. Postopek prepisovanja in zgoščevanja lahko uporabimo za spremembo jezika in oblike zapisa zbirke podatkov, npr. kriptiramo doslej nezaščitene podatke. Metoda ne prepoveduje uporabe enakega imena datoteke za izvor in cilj, vendar to ni priporočljivo. Če bi med prepisovanjem prišlo do razpada sistema, bi lahko izgubili vse podatke. Uporaba drugega imena je priporočljiva tudi zaradi sprotne izdelave rezervne kopije, kakor nam ponazarja naslednji zgled:

```
If Dir("Rezerva.MDB") <> "" Then Kill "Rezerva.MDB"
CompactDatabase "Zbirka.MDB", "Rezerva.MDB", , dbEncrypt
Kill "Zbirka.MDB"
FileCopy "Rezerva.MDB", "Zbirka.MDB"
```

Na koncu imamo na disku kriptirano in zgoščeno obliko zbirke podatkov, ter njeno najbolj posodobljeno rezervno kopijo.

Popravljanje

Nekatere nepopolne ali nepravilne podatke v zbirki je mogoče popravljati s klicem metode **RepairDatabase**, ki prav tako pripada predmetu DBEngine. Podobno kakor zgoščevanje lahko tudi popravljanje uporabimo le za zaprto zbirko podatkov in ga ne moremo vključiti v transakcije. Metoda RepairDatabase prejme le en parameter, ime zbirke podatkov:

```
[DBEngine.]RepairDatabase zbirka
```

Metoda ne prepisuje podatkov v novo zbirko, zato lahko popravljanje poveča velikost datoteke. Po vsakem popravljanju je priporočljivo uporabiti še zgoščevanje.

Vsebina

Doslej smo predmete modela DAO uporabljali za izgradnjo in vzdrževanje strukture zbirke podatkov (*data definition*). V nadaljevanju si bomo ogledali predmete za delo s samimi podatki v zbirki (*data manipulation*). Tudi za obravnavanje podatkov imamo na voljo skoraj prebogat nabor predmetov, metod in lastnosti, ki omogoča morda že preveliko prilagodljivost.

Razred Recordset

S predmetom razreda **Recordset** smo se že zelo podrobno seznanili, ko smo obravnavali gradnik Data. Njegova istoimenska lastnost je pravzaprav samodejno vzpostavljeni predmet tega razreda, ki ga določajo druge lastnosti gradnika Data. Kadar se z zbirkami podatkov spopademo izključno iz programske kode, imamo na voljo razred Recordset, katerega izvode (torej predmete) lahko poljubno ustvarjamo in uporabljamo enako kakor omenjeno lastnost.

Vrste

Predmet razreda Recordset predstavlja zbirko zapisov iz zbirke podatkov, ki je lahko neposredna slika tabele ali pa izbor zapisov iz več tabel oziroma izid poizvedbe. Ko smo uporabljali gradnik Data, smo imeli na voljo tri različne vrste zbirk zapisov. Če predmete razreda Recordset ustvarjamo programsko, pa jih imamo na voljo pet. Opisani so v spodnji tabeli:

Vrsta zbirke zapisov	Opis
Table	Zbirka zapisov vrste Table je neposredna slika tabele v zbirki ali priključene tabele v zunanji *domorodni* zbirki podatkov. Vse spremembe podatkov v zbirki zapisov se odrazijo neposredno v tabeli. Če tabela obsega indekse, je mogoče po zbirki zapisov hitro iskati z metodo Seek. Tovrstno hitro iskanje omogoča samo ta vrsta zbirke zapisov. To je najučinkovitejša zbirka, vendar uporabna le za tabele.
Dynaset	Najzmogljivejša zbirka zapisov je Dynaset. Vsebuje lahko popolno sliko tabele ali izbor zapisov iz ene ali več tabel, ki so izid poizvedbe. Posebej močna zmožnost zbirke vrste Dynaset je uporaba podatkov iz heterogenih tabel, saj omogoča hkratno delo z izborom polj iz vsebovanih in priključenih tabel zbirke podatkov najrazličnejših oblik zapisa. Zbirka Dynaset obsega sklicevanja na dejanske podatke in skrbi za njihovo obnavljanje: vse spremembe se odrazijo v tabelah, spremembe morebitnih drugih uporabnikov na obstoječih zapisih pa se odrazijo v odprti zbirki Dynaset.

Vrsta zbirke zapisov	Opis
Snapshot	Zbirka zapisov vrste Snapshot predstavlja posnetek stanja izbranih zapisov v trenutku, ko je bila ustvarjena. Podobno kakor zbirka Dynaset lahko predstavlja sliko tabele ali izbor zapisov, vendar njene vsebine ni mogoče popravljati[52]. Zaradi bralne narave je uporaba zbirke vrste Snapshot pogosto precej bolj učinkovita od zbirk Table ali Dynaset.
Forward-only	Še posebej učinkovita oblika zbirke vrste Snapshot, ki omogoča samo pregledovanje podatkov v eni smeri. Pri pregledovanju zbirke lahko uporabljamo le metodi MoveNext ali Move s pozitivnim odmikom.
Dynamic	Posebna oblika zbirke, na voljo samo pri virih ODBC. Podobna je zbirki Dynaset.

NAMIG Ker zbirka vrste Snapshot vsebuje popolno kopijo izbranih zapisov, zbirka Dynaset pa se nanje le sklicuje, je lahko slednja pri veliko podatkih učinkovitejša. Tedaj velja razmisliti o uporabi tipa Forward-only. Če ga ni mogoče uporabiti, lahko s preizkusom določimo, katera oblika zbirke zapisov je učinkovitejša.

Za neposredno delo s tabelami je najučinkovitejša zbirka vrste Table. Pri uporabi izbora zapisov iz ene ali več tabel pa si moramo pomagati s kako drugo zbirko. Če želimo podatke samo pregledovati, je najbolje uporabiti zbirko vrste Snapshot, pri enosmernem pregledu pa vrsto Forward-only. Drugače uporabimo zbirko vrste Dynaset.

Uporaba

V programski kodi ustvarimo nov izvod razreda Recordset s klicem metode **OpenRecordset**. Doslej smo spoznali naslednje predmete, ki poznajo to metodo:

- Predmeti razreda Database
- Predmeti razreda TableDef
- Že odprti predmeti razreda Recordset.

Njena skladnja je za rabo z razredom Database naslednja:

```
zbirka.OpenRecordset(vir [, tip] [, opcije] [, zaklepanje])
```

Pri drugih predmetih pa se glasi:

[52] Izjemoma je popravljanje zapisov v zbirki Snapshot mogoče pri nekaterih virih ODBC.

vir.**OpenRecordset**([*tip*] [, *opcije*] [, *zaklepanje*])

Vir predstavlja ime tabele, shranjene poizvedbe ali obstoječega predmeta razreda Recordset. Pri uporabi z zbirko podatkov lahko kot vir predpišemo tudi stavek v jeziku SQL. Prvi parameter metode, tip, predstavlja eno izmed vrst zbirke zapisov. Za vsako je na voljo pripadajoča konstanta:

Vir	Konstanta
Table	dbOpenTable
Dynaset	dbOpenDynaset
Snapshot	dbOpenSnapshot
Forward-only	dbOpenForwardOnly
Dynamic	dbOpenDynamic

Ta parameter ni obvezen, saj so za posamezne vire že na voljo privzeti tipi. Vsaka vrsta ni vedno mogoča, zato si oglejmo različne vrste odpiranja zbirke zapisov:

- Pri odpiranju tabele iz zbirke podatkov so na voljo vse vrste zbirke zapisov, privzeta pa je Table.
- Pri odpiranju kateregakoli drugega elementa zbirke podatkov ali pri uporabi poizvedbe so na voljo zbirke vrst Dynaset, Snapshot in Forward-only. Privzeta vrsta je Dynaset.

Naslednji neobvezni parameter pri odpiranju zbirke zapisov so izbire. Z njimi lahko dodatno določimo vedenje izbrane zbirke zapisov. Še posebej so uporabne konstante, ki določajo vedenje v večuporabniškem okolju. Na voljo so povsem enake konstante, kakor smo jih spoznali, ko smo obravnavali lastnosti Options gradnika Data (glej Gradniki za dostop do podatkov zgoraj). Vedenje v večuporabniškem okolju dokončno določimo z zadnjim neobveznim parametrom zaklepanja. Na voljo so nastavitve:

Zaklepanje	Opis
dbReadOnly	Preprečuje spreminjanje zapisov. Ta nastavitev nadomešča enako nastavitev pri izbirah, ki je tam na voljo le zaradi združljivosti s staro kodo.
dbPessimistic	Določa pesimistično zaklepanje pri popravljanju zapisov.
dbOptimistic	Določa optimistično zaklepanje pri popravljanju zapisov.
dbOptimisticValue	Posebna oblika optimističnega zaklepanja, ki temelji na vsebini in ne na oznaki zapisa. Na voljo le za vire ODBC.

Pri odpiranju zbirke zapisov določimo le začetno nastavitev zaklepanja. Naknadno lahko nanjo vedno vplivamo z lastnostjo LockEdits, ki smo jo že spoznali, ko smo obravnavali gradnik Data.

Oglejmo si zgled odpiranja zbirke zapisov:

```
Dim db As Database
Dim rs As Recordset

  Set db = OpenDatabase("Biblio.MDB")
  Set rs = db.OpenRecordset("Authors", dbOpenDynaset)
  rs.MoveLast
  MsgBox rs.RecordCount ' pravilno število vseh zapisov, okrog 6000

  Set rs = rs.OpenRecordset(dbOpenForwardOnly)
  rs.Move 500
  MsgBox rs.RecordCount ' samo pregledani zapisi: 501
```

Odprli smo tabelo in jo predstavili z zbirko zapisov vrste Dynaset. Z drugim klicem metode smo zbirko »preuredili« v posebno obliko, ki omogoča le pregledovanje naprej. Klic metode MoveLast na njej ni več dovoljen. Premikanje, iskanje in posodabljanje zapisov je pri uporabi predmetov razreda Recordset enako uporabi istoimenske lastnosti gradnika Data, zato ga tu ne bomo znova obravnavali.

Urejanje in izbiranje zapisov

Kadar želimo uporabiti zapise v urejenem vrstnem redu, če je le mogoče, posežemo po tabeli in njenem indeksu. Neposredna uporaba indeksov za urejanje zapisov in hitro iskanje z metodo Seek je mogoča le pri zbirki zapisov vrste Table. Veljavni indeks določimo z lastnostjo Index. Vrstni red zapisov bo nemudoma odrazil urejenost, ki jo ta indeks določa. Oglejmo si zgled:

```
Dim db As Database
Dim rs As Recordset

  Set db = OpenDatabase("Biblio.MDB")
  Set rs = db.OpenRecordset("Titles", dbOpenTable)
  rs.Index = "PrimaryKey"
  rs.MoveFirst
  MsgBox rs!Title ' dBase ...
  rs.Index = "Title"
  rs.MoveFirst
  MsgBox rs!Title ' 1-2-3 ...
  db.Close
```

Če za želeno ureditev indeksa še ni, ga lahko dodamo iz programske kode:

```
Dim db As Database
Dim rs As Recordset
Dim idx As Index
Dim fld As Field
```

```
Set db = OpenDatabase("Biblio.MDB")
Set idx = db.TableDefs("Titles").CreateIndex("Year")
Set fld = idx.CreateField("Year Published")
fld.Attributes = dbDescending
idx.Fields.Append fld
db.TableDefs("Titles").Indexes.Append idx

Set rs = db.OpenRecordset("Titles", dbOpenTable)
rs.Index = "Year"
rs.MoveFirst
MsgBox rs!Title ' Visual C++ ...

db.Close
```

Dodali smo indeks, ki naslove uredi po letnicah padajoče, tako da so najnovejše knjige pri vrhu.

Pri uporabi drugih vrst zbirk zapisov ponavadi urejanje določimo že s poizvedbo, ki jih določa. Začasna rešitev je lahko tudi nastavitev lastnosti **Sort**, ki jo omenjamo le zato, ker se doslej z jezikom SQL še nismo podrobneje seznanili. Podobna opazka velja tudi za lastnost **Filter**, s katero lahko izsejemo zapise, ki ustrezajo določenemu merilu. Mnogo bolje in učinkoviteje je uporabiti merilo že v poizvedbi, ki zbirko zapisov ustvari. Vseeno si oglejmo kratek zgled uporabe omenjenih lastnosti:

```
Set rs = db.OpenRecordset("Titles", dbOpenSnapshot)
rs.Filter = "Title < 'C'"
rs.Sort = "-[Year Published], Title"
Set rs = rs.OpenRecordset
MsgBox rs!Title  ' 3D Rendering ...
```

V zgornjem zgledu smo izločili le naslove, katerih naslov se začne z znakom, »manjšim« od črke C (cifre ter črki A in B), in jih uredili po letnicah *padajoče* (zato predznak minus, -) in po naslovu. Nastavitve lastnosti Filter in Sort učinkujejo šele pri vnovičnem odpiranju zbirke. Dosežemo ga tako, da na že določenem predmetu vnovič izvedemo metodo OpenRecordset.

Prepisovanje zapisov

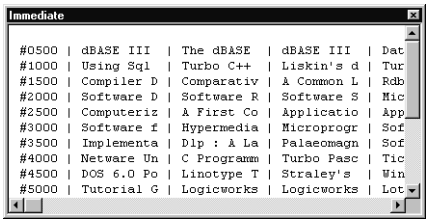
Če želimo določene podatke iz zbirke prepisati v pomnilnik, je najuporabnejša podatkovna struktura dvodimenzionalno polje (*array*). Lahko bi prepisovali vsako posamezno polje, zapis za zapisom, na voljo pa imamo tudi precej hitrejši način. Metoda **GetRows** prebere vsa polja poljubnega števila zapisov in ga vrne v obliki dvorazsežnega polja znotraj prilagodljivega tipa Variant.

Oglejmo si zgled:

```
Dim db As Database
Dim rs As Recordset
Dim vnt As Variant
Dim i As Integer
```

```
Set db = OpenDatabase("Biblio.MDB")
Set rs = db.OpenRecordset("Titles", dbOpenDynaset)
With rs
  Do Until .EOF
    vnt = .GetRows(500)
    ' Sploh vrnjen kak zapis?
    If UBound(vnt, 2) > 0 Then
      Debug.Print
      Debug.Print "#" & Format(.AbsolutePosition, "0000");
      ' Izpišemo vsako polje
      For i = 0 To UBound(vnt, 1)
        Debug.Print " | " & Left(vnt(0, i) & Space(10), 10);
      Next
    End If
  Loop
End With
db.Close
```

Slika 12-6: Izpis prepisanih zapisov



Metoda GetRows vrne želeno število zapisov od dejavnega zapisa dalje. Če vmes naleti na konec zbirke zapisov, vrne manjše število zapisov. Branje hkrati tudi prestavi dejavni zapis za prebrano število zapisov naprej.

Zaporedne številke prve razsežnosti polja določajo polje zapisa oziroma stolpec tabele, druga razsežnost pa posamezen zapis oziroma vrstico tabele. S preverjanjem druge razsežnosti smo v zgornjem zgledu ugotavljali, ali je bil sploh še vrnjen kak zapis. Zanka, ki se sprehodi po prvi razsežnosti, pa nam je omogočila izpis vrednosti posameznih polj.

> **NAMIG** Ker metoda GetRows vedno vrne vsa polja zapisov, je nevarno, da bi hitro porabili pomnilnik, saj so v tabeli lahko zapisi posebnih podatkovnih tipov, ki lahko obsegajo gigantske količine podatkov. Praviloma metodo GetRows uporabimo na zbirki zapisov, ki je izid poizvedbe z natančno določenimi polji, ki jih obsega.

Podatkovni tipi v zbirki

Polju zapisa je mogoče prirediti kar 21 različnih podatkovnih tipov, ki jih podpira predmetni model DAO. Večina jih ustreza osnovnim podatkovnim tipom visual basica, nekateri pa so posebej prirejeni določenim vsebinam, ki se lahko pojavljajo

v zbirkah podatkov. Nekatere posebneže si bomo ogledali v nadaljevanju. Vsi podatkovni tipi so na voljo le znotraj domorodnih zbirk podatkov. Pri zbirkah drugih oblik zapisa poizkuša podatkovni stroj Jet uporabiti najboljši razpoložljivi približek. Podobno je tudi pri zbirkah podatkov, ki jih ustvarimo s stavki v jeziku SQL, in pri oddaljenih virih ODBC, pri katerih je fizični zapis podatkov v celoti odvisen od podatkovnega strežnika.

Tip Long

Uporabo tipa Long (konstanta dbLong) za enolično določen števec, primeren za primarni ključ, smo že spoznali pri zgledu vzpostavitve zbirke podatkov. Na splošno je ta podatkovni tip sposoben shraniti cela števila v enakem obsegu kakor enakovreden tip v visual basicu.

Tip Text

Tip Text (konstanti dbChar in dbText) je edini, ki zahteva tudi predpis velikosti. Velikost mora biti ustrezna najdaljšemu možnemu besedilu, ki se lahko pokaže v tem polju. Na voljo je od 1 do 255 znakov. Če velikosti ne predpišemo ali če uporabimo velikost 0, se privzame največja dolžina. S posebnim atributom lahko določimo, da se v zbirki podatkov dolžina polja dinamično prilagaja vsebini, in tako zmanjšamo prostorske zahteve (ta atribut je uporaben predvsem pri zelo spremenljivi dolžini vsebine in velikem številu zapisov). Za besedila, daljša od 255 znakov, moramo uporabiti poseben tip Memo.

Tipa Memo in LongBinary

Tip Memo (dbMemo) in tip LongBinary (dbLongBinary) lahko – če določena oblika zapisa to dopušča – shranita več kakor 1 GB podatkov. Zato njuno vsebino imenujemo tudi *obsežna vrednost*. Obsežna vrednost (*large value*) je v zbirki zapisana na poseben način, ki kar najmanj vpliva na učinkovitost rabe preostalih podatkov. Polja z obsežnimi vrednostmi ne moremo vključiti v indeks.

Omenjena tipa uporabljamo predvsem za shranjevanje daljšega besedila in večpredstavne vsebine (npr. bitnih slik), velikokrat v obliki predmetov. Dolgo besedilo lahko pregledujemo z večvrstičnim vnosnim poljem, druge vsebine pa z vsebnikom OLE, katerega rabo smo že spoznali.

Pri uporabi obsežne vrednosti iz programske kode nam pride prav posebna lastnost **FieldSize**. Vrne število zlogov, ki jih zaseda vsebina polja. Del obsežne vsebine lahko preberemo z metodo **GetChunk**, v polje pa jo dodamo z metodo **AppendChunk**. Naslednji zgled prepiše obsežno vsebino med dvema zapisoma:

```
Private Sub Prepis(fld1 As Field, fld2 As Field)
Const ChunkLen = 32768
Dim lngOdmik As Long
```

```
Dim lngVelikost As Long
Dim strChunk As String

  ' Prepisujemo obširno vrednost v kosih po 32K zlogov
  lngVelikost = fld1.FieldSize
  Do While lngOdmik < lngVelikost
    strChunk = fld1.GetChunk(lngOdmik, ChunkLen)
    fld2.AppendChunk strChunk
    lngOdmik = lngOdmik + ChunkLen
  Loop

End Sub
```

Tip GUID

Tip GUID omogoča shranjevanje posebne vrednosti, globalno enoličnega označevalca (*Globally Unique Identifier*), ki je v rabi pri oddaljenih klicih, povezovanju odjemalca in strežnika, označevanju replik itn. Nabor 256 enoličnih GUID je mogoče rezervirati pri podjetju Microsoft, mogoče pa jih je tudi izračunati iz enolične oznake omrežne kartice.

Zveza z gradniki

Delo z zbirkami podatkov smo začeli obravnavati pri vezanih gradnikih. Njihova uporaba v marsičem olajša delo s podatki, vendar ne ponuja vse prilagodljivosti uporabe predmetov modela DAO. Če imamo katero od boljših različic visual basica (Professional ali Enterprise), je mogoče dobre strani obeh pristopov tudi kombinirati:

- Zbirko zapisov, ki jih predstavlja gradnik Data, lahko prepišemo v predmet razreda Recordset.
- Poljubno zbirko zapisov, ki jo izdelamo iz programske kode, lahko priredimo gradniku Data skozi lastnost Recordset.

Predvsem slednje je zelo uporabno za izkoriščanje zmogljivosti gradnikov, ki jih moramo navezati na gradnik Data. To postorimo med načrtovanjem vmesnika, določanje vira podatkov pa lahko v celoti obdelamo v programski kodi.

Za zgled si pripravimo uporabniški vmesnik, ki uporablja zunanji gradnik Microsoft Data Bound Grid ter gradnika Data in ukazni gumb z naslednjimi lastnostmi:

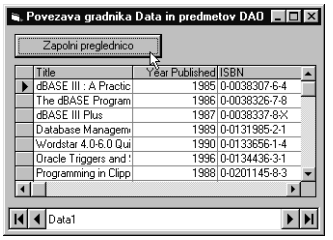
Gradnik	Lastnost	Nastavitev
ukazni gumb (*CommandButton*)	(Name)	Command1
	Caption	»Zapolni preglednico«
podatki (*Data*)	(Name)	Data1
	Align	2 – Align Bottom

Gradnik	Lastnost	Nastavitev
vezana preglednica (*DBGrid*)	(Name)	DBGrid1
	DataSource	Data1

Med načrtovanjem nismo določali nobenih lastnosti gradnika Data, ki bi opredelile podatkovni vir. Omejili smo se le na vzpostavitev povezave med gradnikom Data in vezano preglednico. Zapolnitev preglednice opravi naslednja odzivna procedura:

```
Private Sub Command1_Click()
Dim db As Database

  Set db = OpenDatabase("Biblio.MDB")
  Set Data1.Recordset = db.OpenRecordset("Titles")

End Sub
```

Slika 12-7: Kombiniranje predmetov DAO in vezanega gradnika



Transakcije

Domorodne zbirke podatkov in večina virov ODBC podpira *transakcije*. Transakcija (*transaction*) je zaporedje operacij nad zapisi v zbirki podatkov, ki jo na koncu lahko v celoti izvedemo ali pa zavržemo in s tem izničimo vse spremembe. Transakcijo ponavadi uporabimo za zaporedje soodvisnih operacij, ki jih je treba izvršiti v celoti ali pa sploh ne. Pogosto v tej zvezi omenjamo zgled bančnega avtomata. Ta ne sme izplačati denarja in posodobiti svojega stanja, dokler ne obremeni računa uporabnika. Vse zahtevane operacije združimo v transakcijo, ki jo lahko v celoti prekličemo, če katerakoli izmed operacij spodleti.

Ko zbirko podatkov odpremo, je odprta v stanju *takojšnjega popravljanja* (*auto commit*). To pomeni, da se vse morebitne spremembe po klicu metode Update zapišejo neposredno v ustrezne tabele zbirke podatkov. Tako vedenje je lahko dostikrat neustrezno, saj so morda popravki med seboj odvisni. Še huje je, če med več zaporednimi popravki pride do nepričakovane usodne napake ali, denimo, izpada elektrike. Tedaj imamo v zbirki nekaj posodobljenih in nekaj zastarelih podatkov.

Te težave uspešno rešujejo transakcije. Uporabo transakcij omogočajo tri metode delovnega prostora, predmeta razreda Workspace. Pred začetkom zaporedja

pomembnih operacij pokličemo metodo **BeginTrans**. S tem določimo, da se povzročene spremembe shranjujejo ločeno. Na koncu lahko pokličemo metodo **CommitTrans**, ki transakcijo izpelje do konca in vse popravke trajno zapiše v zbirko podatkov, ali pa metodo **RollBack**, ki celotno transakcijo prekliče in s tem zavrže vse popravke. Omogočeno podporo transakcijam določa lastnost **Transactions**, ki jo pozna predmet razreda Database in vse zbirke zapisov (*recordsets*). Oglejmo si zgled:

```
Private Sub VarnaTransakcija(rs As Recordset)

  If rs.Transactions Then

    BeginTrans

    ' kritične operacije

      BeginTrans

      ' še več kritičnih operacij

      If pogoj1 Then
        CommitTrans
      Else
        Rollback
      End If

    If pogoj2 Then
      CommitTrans
    Else
      Rollback
    End If

  End If

End Sub
```

Kakor vidimo, lahko transakcije gnezdimo. Zgoraj nakazani nabor kritičnih operacij se bo izvedel le, če sta resnična tako prvi kakor drugi pogoj. Skupaj je na voljo največ pet nivojev gnezdenja, pri čemer moramo vsak nivo bodisi končati bodisi preklicati, preden se vrnemo v nadrejenega. Metode za nadzor transakcij v privzetem delovnem prostoru – `DBEngine.Workspaces(0)` – lahko zapišemo neposredno, brez sklicevanja nanj, kar smo storili tudi v zgornjem zgledu.

Transakcije vplivajo na vse predmete v dejavnem delovnem prostoru. Tudi če smo med transakcijo spreminjali več zbirk podatkov, končna metoda vpliva na vse spremembe. Podatkovni stroj nikoli implicitno ne potrdi transakcije. Če v odprti transakciji zapremo delovni prostor, zbirko podatkov ali podprogram, v katerem sta njima pripadajoči predmetni spremenljivki krajevno napovedani, se transakcija samodejno prekliče.

Podatkovni stroj Jet nadzira transakcije tako, da vse spremembe zapisuje v vmesno zbirko podatkov. Zanjo uporablja vmesni hitri pomnilnik (*cache*), ki pa ga lahko hitro zmanjka. Tedaj začne spremembe zapisovati v začasno datoteko na disku, ki jo ustvari v mapi začasnih datotek (npr. c:\windows\temp, odvisno od nastavitve sistemske spremenljivke TEMP). Če zmanjka še prostora na disku, se sproži

izvajalna napaka. Pri popolnoma varnih transakcijah moramo to napako prestreči in se nanjo odzvati s preklicem celotne transakcije.

> **NAMIG** Pri starejših različicah predmetnega modela DAO je bilo mogoče z uporabo transakcij znatno pospešiti izvajanje zaporednih obdelav podatkov. Pri predmetni knjižnici DAO 3.5 je tako vedenje vgrajeno v sam podatkovni stroj.

Delovni prostori

Z delovnimi prostori (*workspaces*) lahko vzpostavljamo poimenovane in varne *seje*. Seja (*session*) je zaporedje operacij v zbirki podatkov, ki se začne s prijavo (*log on*) in konča z zaprtjem delovnega prostora ali odjavo. Vse operacije v zbirki podatkov, ki jih ne uvrščamo v posebno sejo, se izvajajo v privzetem delovnem prostoru, ki je vedno na voljo. Za uporabo posebnih zmogljivosti delovnega prostora imamo na voljo dvoje:

- Posodobimo lastnosti privzetega delovnega prostora.
- Odpremo nov delovni prostor.

Delovni prostor določa izvod razreda **Workspace**, vsi skupaj pa so zbrani v zbirki Workspaces. Vsak delovni prostor pozna lastne, od drugih delovnih prostorov neodvisne transakcije, omogoča prijavljanje uporabnikov in s tem nadzor nad dostopom do podatkov. Z nastavitvijo lastnosti delovnega prostora oziroma z ustvarjanjem novega lahko omogočimo neposreden dostop do virov ODBC. Uporabo različnih delovnih prostorov prikazuje naslednji zgled:

```
Dim ws As Workspace
Dim wsJet As Workspace
Dim wsODBC As Workspace
Dim prp As Property

  ' Ustvarimo nov delovni prostor,
  ' ki uporablja podatkovni stroj Jet
  Set wsJet = CreateWorkspace("JetWorkspace", "admin", "", dbUseJet)
  Workspaces.Append wsJet

  ' Delovni prostor za neposredni
  ' dostop do podatkovnih virov ODBC
  Set wsODBC = CreateWorkspace("ODBCWorkspace", "ime", "geslo", dbUseODBC)
  Workspaces.Append wsODBC

  ' Sprehod po lastnostih vseh delovnih prostorov
  For Each ws In Workspaces
    With ws
      Debug.Print "Lastnosti prostora " & .Name
```

```
      For Each prp In .Properties
        On Error Resume Next
        If prp <> "" Then Debug.Print Tab; prp.Name & " = " & prp
        On Error GoTo 0
      Next prp
    End With
  Next

  wsJet.Close
  wsODBC.Close
```

Za vsak delovni prostor moramo napovedati drugo predmetno spremenljivko. Ustvarjanje novih delovnih prostorov zahteva posredovanje imena in gesla. Za delovne prostore, ki uporabljajo storitve podatkovnega stroja Jet, sta na začetku na voljo uporabniški imeni »Admin«, »Guest« s praznim geslom. Uporabniško ime »Admin« omogoča neomejen dostop do zbirke. Pri virih ODBC določimo privzeto uporabniško ime in geslo za poznejše navezovanje na podatkovni vir.

Zgornji zgled uporablja za izpis takojšnje okno razhroščevalnika, v katerem lahko preberemo naslednje vrednosti:

```
Lastnosti prostora #Default Workspace#
              Name = #Default Workspace#
              UserName = admin
              IsolateODBCTrans = 0
              Type = 2
Lastnosti prostora JetWorkspace
              Name = JetWorkspace
              UserName = admin
              IsolateODBCTrans = 0
              Type = 2
Lastnosti prostora ODBCWorkspace
              Name = ODBCWorkspace
              UserName = admin
              LoginTimeout = 20
              DefaultCursorDriver = -1
              Type = 1
```

Temeljni predmet podatkovnega modela DAO, predmet DBEngine, pozna tri lastnosti, ki določajo vzpostavitev privzetega delovnega prostora – `Workspaces(0)`, ki se samodejno ustvari, kadarkoli pokličemo katero od metod za delo s podatki. Njihova uporaba je prikazana z naslednjim zgledom:

```
DBEngine.DefaultType = dbUseODBC
DBEngine.DefaultUser = "Uporabnik"
DBEngine.DefaultPassword = "Geslo"

' ustvari se delovni prostor za dostop do vira ODBC
' z zgoraj določenim imenom in geslom
Set ws = DBEngine.Workspaces(0)
```

Več o virih ODBC v nadaljevanju.

Varne seje

Podatkovni stroj Jet pozna vrsto varnostnih mehanizmov, ki omogočajo nadzorovanje dostopa do posameznih podatkov v zbirki. Ti mehanizmi se vzdržujejo s posebno, sistemsko zbirko podatkov. Njeno ime in mapa, v kateri je, sta določena v sistemskem registru, lahko pa ju določimo tudi programsko. Privzeta nastavitev imena sistemske zbirke podatkov je System.MDB brez poti, kar pomeni, da podatkovni stroj Jet išče zbirko v mapi na dejavni poti.

Sistemska zbirka podatkov pripada določeni *delovni skupini*. Delovna skupina (*workgroup*) določa uporabnike in uporabniške skupine, ki sodijo skupaj. Žal v visual basicu ni mogoče ustvarjati novih sistemskih zbirk s podatki o delovni skupini. Pomagati si moramo z orodjem Microsoft Access, lahko pa uporabimo tudi vnaprej pripravljene sistemske zbirke podatkov.

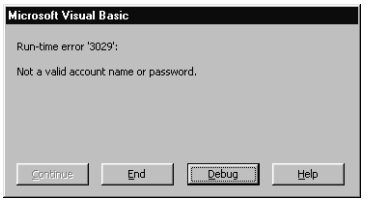
Za demonstracijo uporabe varnostnih mehanizmov je na plošči CD-ROM, priloženi tej knjigi, v osnovni mapi datoteka Varnost.MDW, z osnovno definicijo delovne skupine[53]. Čeprav novih sistemskih zbirk z visual basicom ni mogoče ustvarjati, pa lahko obstoječe popolnoma nadzorujemo, zato nam ena, prototipna zbirka zadošča za nadaljnje delo.

Delovna skupina

Kakor smo videli v dosedanjih zgledih, imamo popolno svobodo pri delu z zbirkami podatkov brez posebnega prijavljanja. Če že odpiramo nove delovne prostore, je na voljo uporabniško ime »Admin« brez gesla, ki ne vsiljuje nobenih omejitev pri delu z zbirkami podatkov. To je privzet, nezavarovan način dela.

Vse se spremeni, če omogočimo sistemsko zbirko podatkov z opisom delovne skupine, ki za uporabnika »Admin« zahteva vnos gesla. To hitro dosežemo tako, da prepišemo datoteko Varnost.MDW v datoteko System.MDB v osnovni mapi visual basica. Če nismo spreminjali nastavitev v registru, bi sedaj podatkovni stroj Jet moral to datoteko najti in jo uporabiti za vsiljevanje varnostnih omejitev. Poizkus kakršnegakoli dela z zbirko podatkov povzroči neuspeh v obliki izvajalne napake:

Slika 12-8: Dejavna zaščita



[53] Čeprav gre za običajno zbirko podatkov v obliki Microsoft Access, je izbran podaljšek .MDW, ki dodatno označuje, da je v zbirki opis delovne skupine.

Namesto da se zanašamo na nastavitve v registru, lahko posežemo tudi po posebni lastnosti predmeta DBEngine, ki določa pot in ime sistemske zbirke in povozi nastavitev v registru. To je lastnost **SystemDB**.

Pomembno: naslednji zgledi zahtevajo, da smo v osnovno mapo visual basica prepisali datoteko Varnost.MDW, ki jo najdemo v osnovni mapi priložene plošče CD-ROM.

Oglejmo si zgled, ki uporabi nastavitve v sistemski tabeli Varnost. V njej je za uporabnika »Admin« določeno geslo »geslo«.

```
Dim ws As Workspace
  DBEngine.SystemDB = "Varnost.MDW"
  Set ws = CreateWorkspace("Delovni prostor", "Admin", "geslo", dbUseJet)
  ' delovni prostor, ki omogoča neomejen dostop do zbirk
  ws.Close
```

Uporabniki in skupine

Znotraj sistemske zbirke podatkov lahko določimo uporabnike, ki jih je mogoče združiti v skupine. Vsakemu uporabniku in skupini pripada enolična varnostna oznaka SID (*security ID*), ki ga natančno opredeljuje. Podatkovni stroj Jet zgradi varnostno oznako iz dveh elementov:

- Imena uporabnika oziroma skupine.
- Posebne osebne oznake (PID), ki mora biti niz, sestavljen iz 4–20 poljubnih znakov.

Ko se uporabnik uspešno prijavi s svojim imenom in geslom, je prav varnostna oznaka SID tista, ki opredeljuje njegove pravice. To pa lahko povzroči vrzel v varnostnem sistemu. Nekateri privzeti uporabniki in skupine imajo vnaprej določene varnostne oznake. Našteti so v spodnji tabeli:

Uporabnik/Skupina	Posebna oznaka PID
uporabnik Admin	Vedno enaka vrednost za vse sistemske tabele.
uporabnik Guest	Vedno enaka vrednost za vse sistemske tabele.
skupina Users	Vedno enaka vrednost za vse sistemske tabele.
skupina Guests	Vedno enaka vrednost za vse sistemske tabele.
skupina Admins	Posebej zgrajena za vsako sistemsko tabelo.

V programu, ki na začetku vpraša po uporabniškem imenu in geslu, je mogoče hitro vsiliti lažno sistemsko zbirko podatkov. Zadošča, da ustvarimo novo sistemsko tabelo, ki uporabniku »Admin« ne predpisuje gesla, in z njo povozimo obstoječo.

Na srečo ima skupina »Admins« enolično določeno oznako PID in s tem tudi varnostno oznako SID. Rešitev zgoraj opisanega problema je v odstranitvi uporabnika »Admin«. To je mogoče le, če je v skupini »Admins« določen še kak uporabnik, saj ta ne sme ostati prazna. Vsi uporabniki v tej skupini imajo najvišji nivo pravic za delo z elementi zbirk podatkov. Imenujemo jih tudi *privilegirani* uporabniki. Za določanje varnega sistema dostopa moramo poznati vsebino te skupine. Posledica zgoraj opisanih značilnosti osnovnih skupin je tudi ta, da so uporabniki in gostje (skupini »Users« in »Guests«), določeni v katerikoli sistemski zbirki podatkov, vedno pripadniki iste varnostne skupine. Zato je za ti dve skupini smiselno določiti le tiste pravice, ki jih želimo dodeliti vsakomur.

Oglejmo si, kako je mogoče določiti novega uporabnika. Spodnji zgled doseže prav to:

```
Dim ws As Workspace
Dim usr As User

  DBEngine.SystemDB = "Varnost.MDW"

  Set ws = CreateWorkspace("Delovni prostor", "Admin", "geslo", dbUseJet)

  Set usr = ws.CreateUser("Uporabnik", IzracunajPID, "geslo")
  ws.Users.Append usr
  ws.Close
```

Uporabnike dodajamo delovnemu prostoru. Novega določimo z metodo **CreateUser**, ki ji posredujemo uporabniško ime, oznako PID in geslo. Tako določenega uporabnika dodamo v zbirko **Users**. Ti uporabniki še ne pripadajo nobeni skupini, vendar so uporabni. Imena skupin in uporabnikov niso občutljiva za velikost znakov, gesla pa *so*!

Oznako PID bi lahko podali v obliki konstantnega niza, vendar smo se iz varnostnih razlogov odločili za rabo funkcije, ki vnaša več varnosti. Zgled take funkcije bi lahko bil:

```
Private Function IzracunajPID() As String
Static blnRandomize As Boolean
Dim n As Integer, i As Integer

  If Not blnRandomize Then
    Randomize
    blnRandomize = True
  End If

  n = 4 + Int(Rnd * 17)
  For i = 1 To n
    IzracunajPID = Chr(32 + Int(Rnd * 94)) & IzracunajPID
  Next

End Function
```

Z nekaj »naključji« sestavimo niz, dolg od 4 do 20 znakov, kot je potrebno za oznako PID. Ko sta vrednosti PID in gesla določeni, ju ne moremo več prebrati, geslo pa lahko spremenimo le s posebno metodo. Tej metodi moramo posredovati

tudi staro geslo, razen če smo privilegirani uporabnik (npr. iz skupine »Admins«). Oglejmo si zgled, ki spremeni geslo prej določenega uporabnika:

```
ws.Users("Uporabnik").NewPassword "geslo", "novogeslo"
```

Uporabnike je bolj smiselno združevati v skupine. Tako jih lahko hkrati nadzorujemo in jim določamo dostopne pravice. Novo skupino določimo podobno kakor novega uporabnika, le da uporabimo zbirko **Groups**:

```
Dim grp As Group

  Set grp = ws.CreateGroup("Skupina", IzracunajPID)
  ws.Groups.Append grp
```

Ko določamo novo skupino, moramo prav tako določiti njeno ime in oznako PID, gesla pa skupina ne pozna.

V skupino lahko dodajamo samo že ustvarjene uporabnike. Uporabnika najprej »ustvarimo« še za skupino in ga dodamo v njeno zbirko Users:

```
  Set usr = ws.Groups("Skupina").CreateUser("Uporabnik")
  ws.Groups("Skupina").Users.Append usr
```

Obstoječe uporabnike in skupine lahko odstranimo iz zbirk, ki jim pripadajo, z metodo Delete. Oglejmo si še kratek zgled, ki našteje vse določene uporabnike in skupine:

```
  For Each usr In ws.Users
    MsgBox "Delovni prostor -> " & usr.Name
  Next
  For Each grp In ws.Groups
    If grp.Users.Count = 0 Then
      MsgBox "Skupina " & grp.Name
    Else
      For Each usr In grp.Users
        MsgBox "Skupina " & grp.Name & " -> " & usr.Name
      Next
    End If
  Next
```

Pravice in lastnina

Lastne uporabnike in skupine ustvarjamo zato, da jim bomo lahko dodelili različne pravice in jim dodelili v last različne elemente zbirke podatkov. Te lastnosti določamo z zbirko **Containers**, ki opisuje osnovne elemente zbirke podatkov, kot so tabele, obrazci, poročila itn. Iz visual basica so nam dostopne le tabele, poizvedbe in celotna zbirka podatkov.

Ker predmet Container opisuje vse elemente iste vrste, moramo za dostop do posamezne tabele ali poizvedbe uporabiti njegovo zbirko **Documents**. Vsaki tabeli ali poizvedbi pripada ustrezen predmet Document, ki omogoča določanje dostopnih pravic posameznih skupin in uporabnikov ter lastnika elementa. To opravilo je dovoljeno le lastnikom elementa in privilegiranim uporabnikom.

Naslednji zgled ustvari novo zbirko podatkov in v njej tabelo. Novo ustvarjeni elementi zbirke samodejno pripadajo uporabniku, ki jih je ustvaril. Ta lahko določi dostopne pravice vsem drugim skupinam in uporabnikom.

```
Dim db As Database
Dim ws As Workspace
Dim doc As Document

  DBEngine.SystemDB = "Varnost.MDW"
  Set ws = CreateWorkspace("Delovni prostor", "Gazda", "", dbUseJet)

  ' Ustvarjanje zbirke s stavkom v jeziku SQL
  Set db = ws.CreateDatabase("Pravice.MDB", dbLangSlovenian)
  db.Execute "CREATE TABLE Tabela (f1 TEXT (50), f2 LONG)"
  db.Execute "INSERT INTO Tabela (f1, f2) VALUES ('vsebina', 10000)"

  ' Ustrezen predmet Document
  Set doc = db.Containers("Tables").Documents("Tabela")
  ' Obravnavali bomo skupino "Skupina"
  doc.UserName = "Skupina"
  ' Dopustimo posodabljanje zapisov
  doc.Permissions = dbSecReplaceData

  ws.Close
```

> **NAMIG** Zbirko smo ustvarili s stavki v jeziku SQL. Natančneje bomo ta jezik spoznali v naslednjem poglavju.

Lastnost **UserName** določa ime skupine ali uporabnika, za katerega želimo ugotoviti ali nastaviti pravice dostopa. Te določa lastnost **Permissions**. Pri pregledovanju pravic uporabnika nam pride prav še lastnost **AllPermisions**, ki vrne vse pravice določenega uporabnika, vključno s tistimi, ki jih je podedoval iz skupine. Različni elementi zbirke podatkov imajo drugače določene konstante dostopnih pravic. Med seboj jih lahko združujemo s seštevanjem ali logičnim operatorjem Or. Vse so zbrane in opisane v sprotni pomoči.

Pravice je mogoče določati tudi za samo zbirko podatkov, kar nam pride še posebej prav pri sistemski zbirki podatkov. Naslednji zgled predpostavlja, da imamo uporabnika »Upravnik«, ki mu želimo omogočiti popolne pravice pri delu z zbirkami podatkov:

```
Dim con As Container

  Set db = OpenDatabase("Varnost.MDW")
  Set con = db.Containers("Databases")
  con.UserName = "Gazda"
  con.Permissions = dbSecDBAdmin
```

Poleg pravic lahko spreminjamo tudi lastnika izbranega elementa. Po privzeti nastavitvi je lastnik tisti uporabnik, ki je določen element ustvaril. Lastnik si vedno lahko sam popravi pravice, ki jih ima pri dostopu do svojega elementa, zato pri

njegovi rabi praktično nima omejitev. Lastnika spremenimo podobno kakor pravice dostopa, le da uporabimo lastnost **Owner**:

```
Set doc = db.Containers("Tables").Documents("Tabela")
doc.Owner = "Skupina"
```

Tudi nastavitev te lastnosti je lahko ime uporabnika ali ime skupine.

Zaščita zbirke

Še tako dobro vzpostavljen sistem zaščite je lahko zelo ranljiv. Določeni namenski programi, ki uporabljajo podatkovni stroj Jet (npr. Microsoft Access), lahko uporabljajo svoje sistemske zbirke podatkov z opisom delovne skupine in s tem skoraj izničijo našo zaščito.

Na srečo rešitev te težave ni težka. Oblika zapisa domorodne zbirke podatkov omogoča enkripcijo podatkov in zaklepanje same zbirke s posebnim geslom.

> **NAMIG** Preden se odločimo zaščititi zbirko podatkov z geslom, poskrbimo, da gesla ne bomo pozabili. Če ga pozabimo, so podatki v zbirki izgubljeni za vse.

Enkripcijo zbirke podatkov lahko določimo že pri njeni stvaritvi, naknadno pa je mogoče vsako zbirko enkriptirati tudi pri njenem zgoščevanju (*compacting*). Te postopke smo si že ogledali pri opisu vzdrževanja zbirke. Z enkripcijo preprečimo možnost razmeroma enostavnega »prebiranja« vsebine zbirke z urejevalniki besedil.

Geslo za zbirko podatkov določimo podobno, kakor ga spremenimo za uporabnika. Uporabimo metodo NewPassword. Geslo lahko spremenimo le za zbirko, ki jo odpremo v nedeljenem načinu:

```
' Odpiranje v nedeljenem načinu
Set db = ws.OpenDatabase("Pravice.MDB", True)
' Določanje gesla, starega gesla ni bilo,
' zato je prvi parameter prazen niz
db.NewPassword "", "geslo"
db.Close
```

Zbirke sedaj ni mogoče odpreti, ne da bi vnesli geslo. Podamo ga znotraj parametra Connect – ali istoimenske lastnosti, če zbirko odpiramo z gradnikom Data – z določilom PWD:

```
Set db = ws.OpenDatabase("Pravice.MDB", False, False, ";PWD=geslo")
```

Pri uporabi metode OpenDatabase za domorodne datoteke, zaščitene z geslom, moramo našteti vse parametre in ustrezno določiti popoln niz za lastnost Connect. Ta se mora vedno začeti z opisom oblike zapisa, ki je za domorodne zbirke kar prazen niz. Zato se nastavitev parametra začne s podpičjem (;), ki mu sledi določilo gesla.

NAMIG Celotno programsko kodo zgledov, ki se ukvarjajo z zaščito podatkov, pa tudi vse druge zglede iz tega poglavja, najdete na priloženi plošči CD-ROM v mapi \Primeri\3-Podatki\1-Predmetni model\.

Naj še enkrat poudarim, da zgledi iz obravnave zaščite ne bodo delovali, če prej ne prepišete datoteke Varnost.MDW iz osnovne mape plošče CD-ROM v mapo visual basica.

13. POGLAVJE

Jezik SQL

Jezik SQL (*Structured Query Language*) je široko veljaven standard za delo z zbirkami podatkov. Z manjšimi razlikami v narečjih ga uporabljajo praktično vsi izdelovalci zmogljivih podatkovnih strežnikov, precej časa pa je že upoštevan tudi v večini manj zmogljivih namenskih programov za delo z zbirkami podatkov. Ker je zelo zmogljiv in uporaben z večino programske opreme za obdelovanje zbirk podatkov, je njegova raba najbolj priporočljiv način dela s podatki tudi v visual basicu.

Podatkovni stroj Jet omogoča uporabo stavkov v jeziku SQL za izdelovanje shranjenih poizvedb, tvorjenje zbirk zapisov, poseganje v strukturo zbirke podatkov, s posebnim določilom pa je mogoče s stavki SQL tudi neposredno ukazovati podatkovnemu strežniku.

NAMIG Jezik SQL je povsem drugačen jezik od narečja basica, uporabljenega v visual basicu, čeprav imata drobne stične točke. Stavke, zapisane v jeziku SQL, zapišemo v nizu znakov, ki ga posredujemo podatkovnemu stroju Jet ali oddaljenemu viru podatkov ODBC.

Raba

Z ukazi v jeziku SQL je mogoče posegati v strukturo zbirke podatkov (*data definition*) in delati s podatki (*data manipulation*). Če zbirko podatkov v celoti nadzorujemo s stavki v SQL, smo s tem razvili precej odprto rešitev, saj je mogoče enake stavke skoraj brez popravkov uporabiti v kateremkoli podatkovnem strežniku. Ker tudi sam podatkovni stroj Jet podpira uporabo SQL, je mogoče razviti prototipne programe, ki uporabljajo podatke iz manj zmogljive zbirke, preizkušeni program pa silno hitro prenesemo v zmogljiv sistem z odjemalci in podatkovnimi strežniki.

Pri delu s podatki nam jezik SQL omogoča relacijski pristop, v katerem z *relacijsko poizvedbo* izsejemo samo tisti del podatkov, ki jih resnično potrebujemo. Pri tem lahko uporabimo vse prednosti relacijske zbirke, kot je združevanje zapisov iz več tabel, njihovo urejanje in omejevanje. Stavki v SQL omogočajo tudi zapis *akcijskih poizvedb*, s katerimi v enem stavku spremenimo, dodamo ali odstranimo poljubno

število zapisov. Za programerje, vajene datotečno usmerjenih orodij, kot so dBase ali FoxPro, terja učinkovita uporaba stavkov v SQL manjši miselni preskok. Namesto »brskalnega« pristopa, pri katerem bolj ali manj pregledujemo vse zapise v tabeli in njihova razmerja z drugimi tabelami, popravke pa v zbirko vnašamo v zankah, ki se sprehajajo po omejeni skupini zapisov, imamo sedaj na voljo vso moč jezika SQL. Kot smo spoznali v prejšnjem poglavju, je še vedno mogoče nasloviti celotno tabelo, po njej hitro iskati in se zaporedno pomikati, vendar tak pristop pri večini problemov, ki jih rešujejo relacijske zbirke, ni potreben. V praksi se izkaže, da dostikrat kombiniramo oba načina. Z učinkovito poizvedbo najprej izsejemo želeno zbirko zapisov, ki jo zatem obdelamo zaporedno.

NAMIG Na vprašanje, kaj je hitrejše, poizvedba v SQL ali zaporedno obdelovanje zapisov, ni enostavno odgovoriti. Večinoma je raba stavkov v SQL hitrejša. Zaporedna obdelava lahko prekosi stavke v SQL le pri uporabi tabel in hitrega preiskovanja z metodo Seek na manjšem številu zapisov. Pri časovno najbolj kritičnih segmentih programa je najbolje izmeriti hitrost obeh pristopov in se na podlagi izida meritve odločiti za način obdelovanja podatkov.

Elementi jezika

Jezik SQL je sestavljen iz *ukazov* (*commands*), *določil* (*clauses*), operatorjev (*operators*), izrazov (*expressions*) in *združevalnih* funkcij (*aggregate functions*). Te elemente združujemo v *stavke* (*statements*), ki omogočajo poseg v strukturo zbirke ali uporabo njenih podatkov. Stavki v jeziku SQL skušajo spominjati na stavke v naravnem jeziku, seveda v angleščini. Oglejmo si zgled:

```
SELECT Name, Telephone FROM Publishers WHERE State = "NY" ORDER BY Name
```

Zgoraj zapisani stavek bi lahko »prevedli« nekako takole:

```
Izberi ime, telefon iz založnikov, kjer je država NY, uredi po imenu.
```

Verjetno lahko brez posebne razlage ugotovimo, kaj poizvedba naredi.

Ločilo elementov jezika je presledek, zato moramo imena tabel, indeksov, polj itn., ki vključujejo presledek, zapisati med oglata oklepaja (`[]`). Povsem enako kakor pri krajšem naslavljanju elementov zbirke v visual basicu.

NAMIG Stavke v jeziku SQL lahko zapišemo z malimi ali velikimi črkami, saj jezik ni občutljiv za velikost znakov. Kljub temu se je uveljavil nasvet, ki priporoča, da elemente jezika zapisujemo z velikimi tiskanimi črkami, imena tabel, indeksov, polj itn. pa zapišemo samo z veliko začetnico.

Ukazi

Ukaze jezika SQL lahko razdelimo na dve skupini:

- Ukazi za določanje strukture zbirke.
- Ukazi za delo s podatki.

Oglejmo si njihov pregled v naslednjih dveh tabelah. Najprej ukazi za določanje strukture zbirke podatkov:

Ukaz	Prevod	Opis
CREATE	Ustvari	Omogoča ustvarjanje novih tabel, indeksov in polj.
ALTER	Spremeni	Omogoča spreminjanje tabele z dodajanjem in spreminjanjem polj.
DROP	Odvrzi	Omogoča odstranjevanje tabel in indeksov.

Ukazi za delo s podatki rabijo za zapis *izbirnih* (*select*-) in *akcijskih* poizvedb (*action query*). To so:

Ukaz	Prevod	Opis
SELECT	Izberi	Omogoča zapis izbirne poizvedbe, ki vrne zbirko zapisov.
UPDATE	Obnovi	Omogoča zapis akcijske poizvedbe za posodobitev izbranih polj in zapisov.
INSERT	Vstavi	Omogoča vstavljanje več novih zapisov z akcijsko poizvedbo.
DELETE	Odstrani	Omogoča odstranjevanje več zapisov z akcijsko poizvedbo.

Določila

Določila dodatno opredelijo vsebino, ki jo želimo spremeniti ali izbrati.

Določilo	Prevod	Opis
FROM	Iz	Določa tabele, iz katerih bomo izbrali zapise.
WHERE	Kjer	Določa pogoje, ki jim morajo zapisi ustrezati.
ORDER BY	Uredi po	Določa ključe urejanja zapisov.
GROUP BY	Združi po	Omogoča združevanje podobnih zapisov v skupine.
HAVING	Vsebuje	Določa pogoje, ki jim morajo ustrezati združene skupine zapisov.

Operatorji in izrazi

Jezik SQL pozna sklicevalni operator, računske, logične in primerjalne operatorje, ki se imenujejo in vedejo podobno kakor njihovi ustrezniki v visual basicu. Oglejmo si najprej primerjalne operatorje:

Operator	Opis/Pomen
=	Je enako?
<	Je manjše?
<=	Je manjše ali enako?
>	Je večje?
>=	Je večje ali enako?
<>	Ni enako?
LIKE	Je podobno vzorcu?
BETWEEN	Je med mejama (vključno)?
IN	Je vsebovan v množici?

Več primerjalnih izrazov lahko združimo v sestavljeni pogoj z logičnimi operatorji. Jezik SQL pozna naslednje:

Operator	Pomen
AND	Logični in, oba združena pogoja morata biti resnična.
OR	Logični ali, vsaj eden izmed združenih pogojev mora biti resničen.
NOT	Logični ne, pogoj mora biti neresničen.

Jezik SQL pozna tudi sklicevalni operator, piko (.). Z njim lahko v poizvedbah določimo pripadnost polja ustrezni tabeli.

Konstantne vrednosti znotraj logičnih izrazov v narečju SQL, ki ga uporablja podatkovni stroj Jet, zapisujemo takole:

Vrsta konstantne vrednosti	Zapis	Zgled zapisa
Celo število	Običajno	1042
Realno število	Kot v programski kodi, decimalno ločilo je pika (.).	10.42
Niz	Med enojnimi ali dvojnimi narekovaji (', ").	'Niz'
Datumska vrednost	Kot v programski kodi, med znakoma *hash* (#).	#5/25/95#

Vrsta konstantne vrednosti	Zapis	Zgled zapisa
Nedoločena vrednost	Uporabimo posebno rezervirano besedo SQL.	`NULL`

Podatkovni stroj Jet nam omogoča, da v izrazih uporabljamo tudi vse *vgrajene* računske operatorje in vse *vgrajene* funkcije visual basica. Če npr. želimo zapisati datum v krajevni obliki, lahko uporabimo funkcijo DateValue. Prioriteta operatorjev je primerljiva s prioriteto operatorjev visual basica. Če želimo vplivati na vrstni red vrednotenja izraza, lahko uporabimo oklepaje.

Združevalne funkcije

Združevalne funkcije uporabimo v izbirnih poizvedbah za skupine zapisov. Vrnejo eno vrednost, ki predstavlja povzetek vseh zapisov v skupini. Pri tem imamo na voljo:

Funkcija	Opis
`COUNT`	Prešteje zapise v skupini.
`SUM`	Sešteje vrednosti polja.
`AVG`	Izračuna povprečje vrednosti v poljih.
`MAX`	Vrne največjo vrednost v skupini polj.
`MIN`	Vrne najmanjšo vrednost v skupini polj.

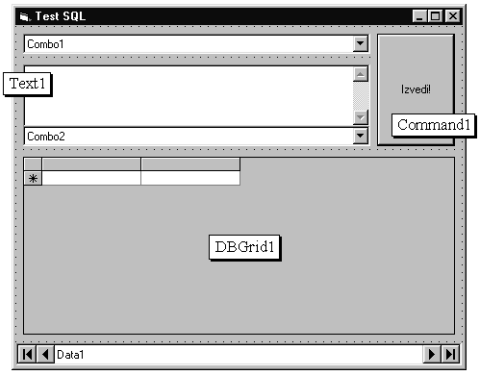
NAMIG Našteti ukazi, določila, operatorji in združevalne funkcije niso vsi elementi jezika SQL. Poznamo še nekatera dodatna določila ter predloge (*predicates*), ki jih bomo spoznali iz zgledov.

Preizkus

Za preizkus praktičnih zgledov v jeziku SQL si bomo pripravili programček, ki bo omogočal vnašanje stavkov, pri izbirnih poizvedbah pa bo tudi prikazal njihov izid. Omogočili bomo delo s tremi zbirkami podatkov, prvo bomo iz nič ustvarili sami, drugi dve pa bosta visual basicu priloženi zbirki podatkov Biblio in Nwind.

Za začetek si pripravimo vmesnik po zgledu s slike 13-1. Potrebovali bomo zunanji gradnik Microsoft Data Bound Grid.

Med načrtovanjem moramo določiti naslednje lastnosti:



Slika 13-1: Vmesnik preizkusnega programa za stavke SQL

Gradnik	Lastnost	Nastavitev
Vnosno polje (*TextBox*)	(Name)	TextBox1
	Font	(Courier, 12 pt)
	Caption	(prazen niz)
Ukazni gumb (*CommandButton*)	(Name)	Command1
	Caption	»Izvedi!«
	Default	True
Zaprti seznam (*ComboBox*)	(Name)	Combo1
	Style	2 – Dropdown List
Zaprti seznam (*ComboBox*)	(Name)	Combo2
	Style	2 – Dropdown List
Podatki (*Data*)	(Name)	Data1
	Align	2 – Align Bottom
Vezana preglednica (*DBGrid*)	(Name)	DBGrid1
	DataSource	Data1
	AllowAddNew	True
	AllowDelete	True

Oglejmo si še programsko kodo, ki bo skrbela za pravilno delovanje zgleda. Najprej poskrbimo za dokončno vzpostavitev vmesnika:

```
Private Const strDBFile = "SQLTest.mdb"
Private db As Database

' Vzpostavitev
Private Sub Form_Load()
  With Combo1
    .AddItem "Zbirka podatkov " & strDBFile
    .AddItem "Zbirka podatkov Biblio.MDB"
```

```
    .AddItem "Zbirka podatkov Nwind.MDB"
    .ListIndex = 0
  End With
  VzpostaviZbirko 0
End Sub
```

Ime datoteke za lastno zbirko podatkov smo zapisali v konstanto. S tem si olajšamo morebitne poznejše popravke. Potrebujemo še modulsko predmetno spremenljivko, ki bo ponazarjala odprto zbirko podatkov. V odzivni proceduri dogodka Load zapolnimo prvi zaprti seznam in pokličemo pomožno proceduro, ki vzpostavi izbrano zbirko:

```
' Pomožna procedura za odpiranje/ustvarjanje zbirke
Private Sub VzpostaviZbirko(intZbirka As Integer)
Static intStaraZbirka As Integer
Dim strZbirka As String

  If (intZbirka + 100) <> intStaraZbirka Then

    intStaraZbirka = intZbirka + 100
    Set db = Nothing
    Select Case intZbirka
      Case 0
        strZbirka = strDBFile
      Case 1
        strZbirka = "Biblio.MDB"
      Case 2
        strZbirka = "Nwind.MDB"
    End Select

    If Dir(strDBFile) = "" Then
      Set db = CreateDatabase(strZbirka, dbLangSlovenian)
    Else
      Set db = OpenDatabase(strZbirka)
    End If

    Data1.DatabaseName = strZbirka
    Data1.RecordSource = ""
    Data1.Refresh

  End If

End Sub
```

Procedura po potrebi ustvari in odpre izbrano zbirko, ki je lahko naša lastna ali že obstoječa Biblio oziroma Nwind. Iz procedure je razvidno, kako obnovimo stanje predmetne spremenljivke in kako posodobimo stanje gradnika Data. Zaradi učinkovitejšega delovanja procedura vedno ve, katero zbirko je odprla nazadnje. Če je uporabnik znova izbral isto zbirko, do vnovičnega izbiranja ne pride. Ker želimo zbirko označiti z enako zaporedno številko, kot je zaporedna številka pripadajoče vrstice na izbirnem zaprtem seznamu Combo1, dobijo zbirke številske oznake 0,1,2. Kakor vemo, so statične spremenljivke na začetku postavljene na privzeto vrednost, zato smo uporabili manjšo zvijačo (prištevanje stotice), da začetno stanje statične spremenljivke določa neveljavno zbirko. Določimo še odzivno proceduro za izbor nove zbirke z zaprtega seznama:

```
' Izbira zbirke
Private Sub Combo1_Click()
  VzpostaviZbirko (Combo1.ListIndex)
End Sub
```

Sedaj smo pripravljeni za uporabo vnesenih poizvedb. Zažene jih klik privzetega ukaznega gumba:

```
' Sprožitev poizvedbe
Private Sub Command1_Click()
Dim strSQL As String
Dim blnNapaka As Boolean
Dim i As Integer

  blnNapaka = False
On Error GoTo napaka:
  strSQL = Trim(Text1.Text)
  If UCase(Left(strSQL, 6)) = "SELECT" _
      And InStr(1, strSQL, "INTO", vbTextCompare) = 0 Then
    With Data1
      .RecordSource = strSQL
      .Caption = strSQL
      .Refresh
    End With
  Else
    db.Execute Text1.Text, dbFailOnError
  End If
  If Not blnNapaka Then
    For i = 0 To Combo2.ListCount - 1
      If UCase(Combo2.List(i)) = UCase(strSQL) Then Exit Sub
    Next
    Combo2.AddItem strSQL, 0
  End If
  Beep
  Exit Sub

napaka:
  blnNapaka = True
  Select Case MsgBox(Err.Description, _
      vbCritical + vbAbortRetryIgnore + vbDefaultButton3, "Napaka!")
    Case vbAbort
      Unload Me
    Case vbRetry
      Resume
    Case vbIgnore
      Resume Next
  End Select
End Sub
```

Pri uporabi stavka SQL imamo vedno dve možnosti:

- Če gre za izbirno poizvedbo, z njo ustvarimo zbirko zapisov.
- Če gre za katerokoli drugo poizvedbo, jo izvedemo z metodo **Execute**.

Mi preverjamo vrsto stavka z ugotavljanjem vsebine niza. Če gre za stavek SELECT, ki pa ne sme vsebovati določila INTO, odpremo pripadajočo zbirko zapisov, drugače stavek izvedemo neposredno. Vidimo, da lahko izbirno poizvedbo v obliki stavka SQL priredimo kar *neposredno* lastnosti RecordSource gradnika Data!

V podprogramu smo vzpostavili tudi prestrezovalnik izvajalnih napak, ki uporabniku prepusti odločitev, kako nadaljevati. Zaradi večje prijaznosti do uporabnika smo uspešne poizvedbe shranili še na drug zaprti seznam, pri katerem omogočimo še prepis v vnosno polje:

```
' Uporaba uspešne poizvedbe
Private Sub Combo2_Click()
  Text1.Text = Combo2.Text
End Sub
```

Na koncu moramo poskrbeti še za zapiranje zbirke podatkov:

```
' Pospravljanje
Private Sub Form_Unload(Cancel As Integer)
  Set db = Nothing
End Sub
```

> **NAMIG** Stavke v jeziku SQL lahko preizkušamo tudi v posebnem oknu orodja Visual Data Manager.
>
> Programsko kodo iz poglavja o jeziku SQL najdete na priloženi plošči CD-ROM v mapi \Primeri\3-Podatki\2-SQL\.

Struktura

Pregled praktične rabe stavkov v jeziku SQL bomo začeli s stavki, ki omogočajo vzpostavitev in spremembo same strukture zbirke podatkov. S temi ukazi lahko ustvarjamo in spreminjamo tabele, indekse, polja in relacije v *domorodnih* zbirkah in oddaljenih virih ODBC, pri katerih imamo ustrezne pravice do dostopa.

> **NAMIG** Strukturo drugih oblik zapisa podatkov, ki jih podpira podatkovni stroj Jet, je mogoče spreminjati s predmeti DAO, seveda v mejah, ki jih določena oblika zapisa dopušča.

Tabele

Novo tabelo ustvarimo s stavkom CREATE TABLE, v katerem predpišemo ime tabele in podamo seznam polj zapisa. Seznam vključuje ime polja, njegov podatkovni tip, če gre za polje z besedilom, pa še njegovo dolžino. Podatkovni tipi

v stavkih SQL so združljivi s tipi, uporabljenimi pri predmetih DAO, vendar se ne imenujejo povsem enako. Njihov pregled je v spodnji tabeli:

Podatkovni tip v SQL	Enakovreden tip v DAO
COUNTER	dbLong z atributom dbAutoIncrField (enolični števec).
BIT	dbBoolean
BYTE	dbByte
CURRENCY	dbCurrency
DATETIME	dbDateTime
SINGLE	dbSingle
DOUBLE	dbDouble
SHORT	dbInteger
INT	dbLong
LONG	dbLong
TEXT (n)	dbText dolžine *n*.
CHAR (n)	dbText dolžine *n*.
VARCHAR (n)	dbText z atributom dbVariableField največje dolžine *n*.
GUID	dbGuid
BINARY	dbBinary
LONGTEXT	dbMemo
LONGBINARY	dbLongBinary

Oglejmo si zgled, ki ustvari tabelo založnikov z imenom »Zalozbe«, kot smo jo določili, ko smo obravnavali predmete DAO:

```
CREATE TABLE Zalozbe ( Zalozba COUNTER, Opis VARCHAR (100) )
```

Naslednji stavek pa ustvari še tabelo knjig:

```
CREATE TABLE Knjige (
  ISBN TEXT (10),
  Naslov VARCHAR (100),
  Avtor VARCHAR (100),
  Zalozba LONG
)
```

Presledki in preostali beli prostor se v stavkih SQL vede le kot ločilo med elementi, zato lahko vsak stavek zapišemo v poljubnem številu vrstic z vmesnimi zamiki, tako

kakor v zgornjem zgledu. S tem znatno izboljšamo berljivost, saj so lahko stavki v jeziku SQL zelo zelo dolgi.

> **NAMIG** V zgledu, ki smo ga razvili za potrebe preizkušanja stavkov v SQL, imamo večvrstično vnosno polje. Zaradi privzetega ukaznega gumba moramo prehod v novo vstico izsiliti s pritiskom na kombinacijo Ctrl+Enter. Pritisk na tipko Enter sproži izvajanje ukaza.

Obe tabeli sta še brez indeksov, vendar smo lahko že zaslutili, kakšno moč ponujajo na videz preprosti stavki jezika SQL. Preden si ogledamo razširitev stavka CREATE TABLE, spoznajmo še nekaj stavkov za spreminjanje strukture zbirke.

Spreminjanje strukture

S stavkom ALTER TABLE je mogoče dodati ali odstraniti poljubno polje zapisa. Z dvema stavkoma lahko hitro spremenimo strukturo tabele. Denimo, da smo ugotovili, da je polje za zapis naslova prekratko. Najprej zavržemo obstoječe polje:

```
ALTER TABLE Knjige DROP COLUMN Naslov
```

Zatem dodamo daljšega:

```
ALTER TABLE Knjige ADD COLUMN Naslov TEXT (150)
```

V žargonu jezika SQL se polje imenuje *stolpec* (*column*), zapis pa *vrstica* (*row*) tabele. Ko odstranimo polje (stolpec), izgubimo tudi njegovo vsebino za vse obstoječe zapise. Če želimo spremeniti strukturo, hkrati pa ohraniti podatke, moramo uporabiti še izbirno poizvedbo, ki podatke shrani, in akcijsko, ki jih vpiše v spremenjeno tabelo.

Odstranjevanje tabel

Neuporabno tabelo zavržemo s stavkom DROP TABLE. Ker bomo v nadaljevanju izdelali z indeksi in relacijo opremljeni različici zgornjih tabel, ju zavrzimo:

```
DROP TABLE Zalozbe
```

In še knjige:

```
DROP TABLE Knjige
```

Celovitost podatkov

Ker je SQL jezik za rabo v relacijskih zbirkah podatkov, je pojem indeksa nekoliko zabrisan. Indekse, relacije in ugotavljanje pravilnosti vnosa obravnavamo kot skupino elementov, ki omogočajo vsiljevanje celovitosti podatkov (*referential integrity*). Zato vse skupaj določa določilo CONSTRAINT (angl. omejitev).

Indeksi

Indeks za izbrano polje ali polja v tabeli lahko dodamo na tri načine:

- Med ustvarjanjem tabele s stavkom CREATE TABLE.
- Med spreminjanjem tabele s stavkom ALTER TABLE.
- S posebnim stavkom CREATE INDEX.

Za dodajanje tujega ključa in vsiljevanje celovitosti podatkov moramo uporabiti prvi dve obliki, zadnja je na voljo izključno za dodajanje indeksov urejanja.

Najpogosteje indekse vzpostavimo že pri ustvarjanju tabele. Tedaj imamo na voljo dve obliki zapisa. Kadar določamo indeks za eno samo polje, ob njegovi napovedi zapišemo še določilo CONSTRAINT:

```
CREATE TABLE Zalozbe (
  Zalozba COUNTER CONSTRAINT Zalozba PRIMARY KEY,
  Opis VARCHAR (100)
)
```

S tem smo določili, da pripada polju »Zalozba« indeks primarnega ključa z enakim imenom. Vsaka tabela ima lahko le en primarni ključ, ki mora vsebovati enolične vrednosti.

Drug način uporabimo pri indeksih, ki združujejo več polj. Tu določilo CONSTRAINT zapišemo kot samostojni element seznama (za vejico):

```
CREATE TABLE Knjige (
  ISBN TEXT (10),
  Naslov VARCHAR (100),
  Avtor VARCHAR (100),
  Zalozba LONG,
  CONSTRAINT [Naslov Avtor] UNIQUE (Naslov, Avtor)
)
```

Določili smo nov indeks z imenom »Naslov Avtor«, ki vsebuje polji Naslov in Avtor, njegova vrednost pa mora biti enolična za vsak zapis.

Če bi bili tabeli že določeni, a brez indeksov, bi lahko posegli po stavku ALTER TABLE, ki omogoča tudi dodajanje in odstranjevanje indeksov. V našem zgledu bomo odstranili obstoječi indeks in ga nadomestili z novim:

```
ALTER TABLE Knjige DROP CONSTRAINT [Naslov Avtor]
```

Sedaj pa dodamo novega, ki je enak prejšnjemu, le da je določen naknadno:

```
ALTER TABLE Knjige ADD CONSTRAINT [Naslov Avtor] UNIQUE (Naslov, Avtor)
```

Zadnji način dodajanja indeksa je uporaba posebnega stavka CREATE INDEX. Gre za skladenjsko spremenjeno obliko zapisa ALTER TABLE ... CONSTRAINT. Naslednji zgled je torej enakovreden prejšnjemu:

```
CREATE UNIQUE INDEX [Naslov Avtor] ON Knjige (Naslov, Avtor)
```

V treh primerih pa nam lahko pomaga samo stavek CREATE INDEX:

- Pri ustvarjanju indeksa za podvajajoče vrednosti. Pri uporabi stavkov CREATE/ALTER TABLE moramo indeksu predpisati primarni ključ ali enolično vrednost polja. To pri uporabi stavka CREATE INDEX ni potrebno:

  ```
  CREATE INDEX Naslov ON Knjige (Naslov)
  ```

  Vzpostavili smo indeks za hitro preiskovanje po naslovih, ki dopušča možnost dveh knjig z istim naslovom. Skupaj s prej vzpostavljenim indeksom še vedno prepovedujemo dve enaki kombinaciji avtorja *in* naslova (kar je res ista knjiga).

- Pri ustvarjanju indeksa za urejanje v padajočem zaporedju (Ž–A, 100–0). Če bi želeli vzpostaviti indeks, ki zapise ureja po padajočih vrednostih ključa, pa uporabimo določilo DESC:

  ```
  CREATE INDEX Naslov ON Knjige (Naslov DESC)
  ```

  Sedaj bo indeks urejal naslove knjig v padajočem zaporedju. Določilo DESC lahko spremlja poljubno število polj na seznamu. SQL pozna tudi določilo ASC, ki pa je privzeto in ga ponavadi ne zapisujemo posebej.

- Pri urejanju zapisov v *priključeni* tabeli, ki še nima vzpostavljenega indeksa. Pri ustvarjanju tovrstnega indeksa ne potrebujemo posebnih pravic pri izvoru podatkov (npr. podatkovnem strežniku), saj se izvorna zbirka podatkov niti ne zaveda, da smo ustvarili indeks. Indeks je shranjen v naši zbirki podatkov in ga lahko uporabljamo kakor vse druge.

Skladnja stavka CREATE INDEX dopušča še uporabo neobveznega določila WITH, s katerim lahko predpišemo tri posebne lastnosti indeksa:

- WITH PRIMARY. Določa indeks primarnega ključa tabele.
- WITH DISALLOW NULL. Onemogoča vnašanje zapisa, pri katerem bi bila vsebina polja nedoločena (*null*).
- WITH IGNORE NULL. Pri izgradnji indeksa dopušča in zanemari polja nedoločene vsebine.

Naslednji zgled preprečuje vnos novega zapisa, pri katerem bi bila vsebina indeksiranega poljaNaslov nedoločena:

```
CREATE INDEX Naslov ON Knjige (Naslov DESC) WITH DISALLOW NULL
```

Relacije

Z določilom CONSTRAINT lahko zapišemo tudi relacijsko povezavo, ki vsiljuje celovitost podatkov. Pojem celovitost podatkov smo spoznali pri delu s predmetom razreda Relation, ki je del predmetnega modela DAO. Tudi pri uporabi jezika SQL veljajo enake značilnosti, zapis pa je seveda popolnoma drugačen.

Najprej si oglejmo splošno, simbolično obliko zapisa določila CONSTRAINT. Kot smo že spoznali, sta obliki zapisa dve. Prva pripada enemu samemu polju:

```
CONSTRAINT indeks
  {
   PRIMARY KEY |
   UNIQUE |
   REFERENCES tujaTabela [(tujePolje1 [, tujePolje2])]
  }
```

Druga oblika je samostojni člen na seznamu elementov tabele in vsebuje več polj:

```
CONSTRAINT indeks
  {
   PRIMARY KEY (ključ1 [, ključ2 [, …]]) |
   UNIQUE (ključ1 [, ključ2 [, …]]) |
   FOREIGN KEY (ključ1 [, ključ2 [, …]])
     REFERENCES tujaTabela [(tujePolje1 [, tujePolje2 [, …]])]
  }
```

V simboličnem zapisu predstavlja *indeks* ime indeksa, $ključ_n$ imena polj ključev, *tujaTabela* ime relacijsko povezane tabele in $tujePolje_n$ ime tujega polja, ki se navezuje na ključ relacije. Vidimo, da je pri obeh oblikah zapisa treba izbrati eno izmed treh možnosti. To so:

- PRIMARY KEY. Določilo označi izbrano polje ali kombinacijo polj kot primarni ključ. Posamezna tabela ima lahko le en primarni ključ, vsaka njegova vrednost pa mora biti enolična za vso tabelo.
- UNIQUE. Označi polje ali kombinacijo polj kot enolično. Polje ali kombinacija vrednosti, če smo izbrali več polj, mora imeti enolično vrednost za vso tabelo.
- FOREIGN KEY/REFERENCES. Označi polje ali kombinacijo polj kot tuji ključ, ki rabi za vzpostavitev relacije s tujo tabelo. V tem primeru moramo določiti še ime tujega polja ali polj, ki določajo relacijo. Če so ta polja v tuji tabeli vsebovana v njenem primarnem ključu, jih ni treba naštevati.

Sedaj lahko zapišemo popolnoma enakovredno obliko stvaritve tabele knjig, kakršno smo ustvarili s predmeti DAO:

```
CREATE TABLE Knjige (
  ISBN TEXT (10) CONSTRAINT ISBN PRIMARY KEY,
  Naslov VARCHAR (100),
  Avtor VARCHAR (100),
  Zalozba LONG CONSTRAINT Zalozba REFERENCES Zalozbe (Zalozba)
)
```

Celovitost zapisa

Ko smo obravnavali predmete DAO, smo omenili obravnavanje določil, ki omogočajo vsiljevanje celovitosti posameznega zapisa. Z nekaj dodatnimi

lastnostmi lahko določimo, da posamezno polje ne sme biti prazno, njegovo privzeto vrednost ob dodajanju novega zapisa in pravilo vnosa, ki dodatno omejuje razpoložljiv nabor vrednosti.

Pri uporabi narečja jezika SQL, ki ga razume podatkovni stroj Jet, nam preostane le eno določilo, ki dodatno določa celovitost polja, to je NOT NULL. Predpišemo ga poljem v stavku CREATE TABLE, za katere *zahtevamo* vnos vrednosti. Prvi kandidat za tako polje je prav gotovo primarni ključ, če ni posebnega tipa COUNTER.

Tako je videti še dodatno dopolnjen zadnji zgled, ki onemogoča vnos novega zapisa z nekaterimi izpuščenimi polji:

```
CREATE TABLE Knjige (
  ISBN TEXT (10) NOT NULL CONSTRAINT ISBN PRIMARY KEY,
  Naslov VARCHAR (100) NOT NULL,
  Avtor VARCHAR (100) NOT NULL,
  Zalozba LONG NOT NULL CONSTRAINT Zalozba REFERENCES Zalozbe (Zalozba)
)
```

Na celovitost zapisa posredno vplivajo tudi indeksi. Primarni ključ ali enolični indeks omejujeta vnos ponavljajočih se vrednosti.

> **NAMIG** Podatkovni strežniki poznajo svoja narečja jezika SQL, ki najpogosteje vsebujejo dodatna določila za določanje celovitosti zapisa (*restrictions*). Tovrstne ukaze lahko uporabljamo tudi v visual basicu, če jih prepustimo neposredno strežniku, brez vmešavanja podatkovnega stroja Jet.

Podatki

Jezik SQL je posebej učinkovit pri izbiranju in posodabljanju vsebine relacijskih zbirk podatkov. Že njegovo ime pove, da je namenjen predvsem poizvedbam (*queries*). Poizvedbe delimo na:

- Izbirne poizvedbe (*select queries*). Tovrstne poizvedbe določa ukaz SELECT. Omogočajo izbor zapisov iz ene ali več tabel. Te poizvedbe vračajo zbirko zapisov.
- Akcijske poizvedbe (*action queries*). Tovrstne poizvedbe določajo ukazi SELECT INTO, INSERT INTO, UPDATE in DELETE. Omogočajo dodajanje, spreminjanje ali odstranjevanje večjega števila zapisov iz ene ali več tabel. Te poizvedbe ne vračajo zapisov.

Najprej si oglejmo preprostejše načine izbiranja podatkov.

Izbor podatkov

Najpogosteje rabljeni stavek v jeziku SQL je stavek SELECT, ki vrne izbor zapisov iz ene ali več tabel. Stavek ne vpliva na njihovo vsebino, vrne zgolj njihovo stanje v trenutku izvajanja poizvedbe. Simbolično lahko zapišemo stavek SELECT takole:

```
SELECT
  [DISTINCT] [TOP n [PERCENT]]
  { * | polje1 [AS ime1][, polje2 [AS ime2][, …]] }
  [ INTO novaTabela ]
  FROM tabela1 [, tabela2 [, …]] [ IN zunanjaZbirka ]
  [ WHERE pogoj ]
  [ GROUP BY sPolje1 [, sPolje2 [, …]] ]
  [ HAVING pogojSkupine ]
  [ ORDER BY uPolje1 [{ASC|DESC}] [, uPolje2 [{ASC|DESC}] [, …]] ]
```

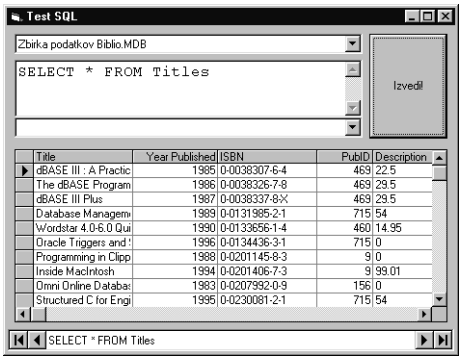
Precej zapleten simbolični zapis, ki pa še vedno ne povzema čisto vseh variacij stavka SELECT.

> **NAMIG** Zgledi, ki bodo sledili, uporabljajo zbirki podatkov Biblio in Nwind. Če za preizkus stavkov v jeziku SQL uporabljate zgled, ki smo ga razvili v tem poglavju, z zgornjega zaprtega seznama izberite »Zbirka podatkov Biblio.MDB« ali »Zbirka podatkov Nwind.MDB«.

Najpreprostejša izbirna poizvedba izbere vse zapise iz navedene tabele. Pri zbirki podatkov Biblio.MDB bi lahko zapisali:

```
SELECT * FROM Titles
```

Posebni simbol, zvezdica (*), označuje vsa polja tabele.



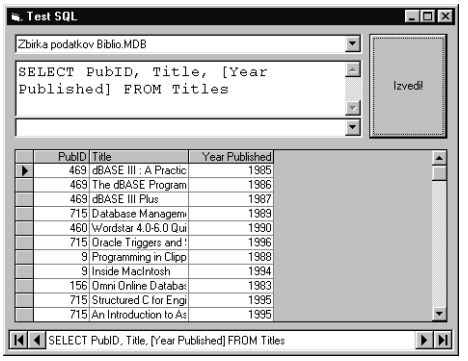
Slika 13-2: Izid poizvedbe po vseh poljih

Čeprav zgornja poizvedba vrne celotno sliko izbrane tabele, njenega izida ne moremo uporabiti kot vrsto zbirke Table. Izid vsake poizvedbe je vedno lahko le zbirka zapisov vrste Dynaset, Snapshot ali ForwardOnly (pri virih ODBC tudi Dynamic).

Če ne želimo vseh polj ali pa želimo zgolj spremeniti njihov vrstni red, zapišemo namesto posebnega znaka seznam polj:

```
SELECT PubID, Title, [Year Published] FROM Titles
```

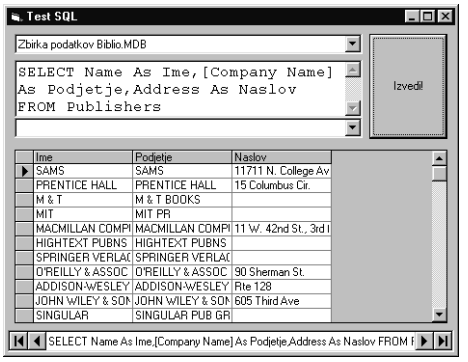
Slika 13-3: Izid poizvedbe po nekaj poljih



Vplivamo lahko tudi na naziv polja v vrnjeni zbirki zapisov. S seznama polj lahko z določilom AS priredimo polju novo ime:

```
SELECT Name As Ime, [Company Name] As Podjetje, Address As Naslov
  FROM Publishers
```

Slika 13-4: Izid poizvedbe s preimenovanimi polji



Izločanje in urejanje

Jezik SQL ponuja več določil in predlogov, s katerimi lahko dodatno izsejemo res samo želene podatke. S predlogom DISTINCT se lahko znebimo podvojenih zapisov. Naslednja poizvedba:

```
SELECT [Year Published] FROM Titles
```

vrne več kakor 8000 zapisov, saj izbere letnice vseh knjig. Med njimi je precej podvojenih. Naslednja poizvedba vrne le 42 različnih letnic, ki jih najdemo pri knjigah:

```
SELECT DISTINCT [Year Published] FROM Titles
```

Dodatno omejitev pri izbiranju zapisov dosežemo z rabo določila WHERE, ki ga spremlja logični izraz. V izid poizvedbe bodo vključeni le tisti zapisi, za katere se izraz izvrednoti v resnično vrednost. Oglejmo si zgled:

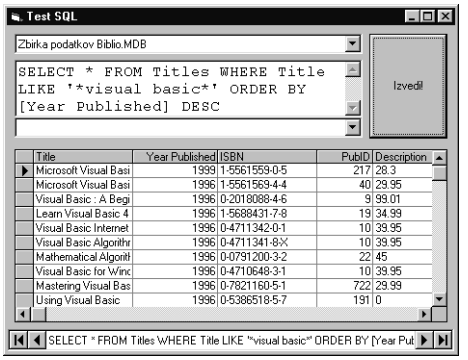
```
SELECT Title FROM Titles WHERE [Year Published] BETWEEN 1980 AND 1989
```

Poizvedba izloči le tiste knjige, ki so bile izdane v 80. letih. Poljem, ki jih uporabimo v zapisu pogoja, ni treba biti tudi v vrnjeni zbirki zapisov.

Podatke je mogoče urejati z določilom ORDER BY. Zapise lahko urejamo po poljubnem merilu, ne glede na to, ali je bil v zbirki vzpostavljen ustrezen indeks ali ne. Morebitni indeksi sicer blagodejno vplivajo na hitrost poizvedovanja. Oglejmo si zgled, ki vrne vse podatke o knjigah o visual basicu, urejene od najnovejše proti najstarejši:

```
SELECT * FROM Titles
  WHERE Title LIKE '*visual basic*'
  ORDER BY [Year Published] DESC
```

Slika 13-5: Poizvedba izsejanih in urejenih zapisov



Narečje jezika SQL, ki ga uporablja podatkovni stroj Jet, pozna še poseben predlog TOP, ki omeji število vrnjenih zapisov. Naslednja poizvedba vrne le prvih 10 knjig o visual basicu glede na vrstni red naslovov:

```
SELECT TOP 10 Title FROM Titles
  WHERE Title LIKE '*visual basic*'
  ORDER BY Title
```

Predlog TOP lahko spremlja še določilo PERCENT. Takole zapisana poizvedba vrne prvih 10 odstotkov knjig o visual basicu (v zbirki Biblio je to 23 naslovov):

```
SELECT TOP 10 PERCENT Title FROM Titles
  WHERE Title LIKE '*visual basic*'
  ORDER BY Title
```

Predlog TOP ne omejuje števila podvojenih vrednosti v polju urejanja zapisov. Če bi zapisali naslednjo poizvedbo:

```
SELECT TOP 10 Title FROM Titles
  WHERE Title LIKE '*visual basic*'
  ORDER BY [Year Published]
```

bi dobili 28 zapisov, ker je precej knjig izšlo v istem letu.

Združevanje zapisov

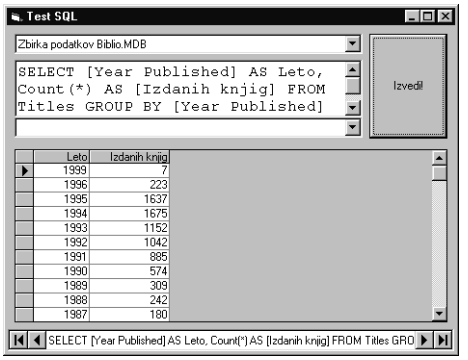
Združevalne funkcije ovrednotijo skupino zapisov in vrnejo njihov povzetek v obliki ene ali več skupin. Verjetno je najpogosteje rabljena združevalna funkcija COUNT, ki vrne seštevek vseh zapisov poizvedbe. Hitro lahko ugotovimo, kolikšno je skupno število knjig v tabeli:

```
SELECT COUNT(*) FROM Titles
```

Združevalni funkciji COUNT lahko kot parameter podtaknemo katerokoli polje zapisa, vendar svoje delo najhitreje opravi, če ji podamo poseben simbol, zvezdico (*). V vrnjeni poizvedbi dobi polje s seštevkom privzeto ime[54], ki pa ga lahko spremenimo z določilom AS. Združevalne funkcije pogosto uporabljamo skupaj z določilom GROUP BY. Tako namesto povzetka vseh zapisov dobimo povzetke posameznih skupin zapisov. Naslednja poizvedba vrne število izdanih knjig, združeno in urejeno po letnici izdaje:

```
SELECT [Year Published] AS Leto, Count(*) AS [Izdanih knjig] FROM Titles
  GROUP BY [Year Published]
  ORDER BY [Year Published] DESC
```

Slika 13-6: Združevanje in povzemanje zapisov



V izbirni poizvedbi z združevalnimi funkcijami lahko nastopajo *le* polja, našteta v določilu GROUP BY. Združevanje ni mogoče po poljih z obsežno vsebino (LongText in LongBinary). Polja nedoločene vrednosti (*null value*) se tudi zberejo v posebno skupino, vendar jih združevalne funkcije ne upoštevajo.

Z določilom WHERE lahko izločimo zapise *pred* združevanjem v skupine. Na voljo imamo tudi določilo HAVING, ki izlušči posamezne skupine *po* združevanju.

[54] Vsa izračunana polja v poizvedbah znotraj podatkovnega stroja Jet dobijo privzeto ime, sestavljeno iz niza »Expr1« in trimestnega zaporednega števila (Expr1000, Expr1001 itn.).

Naslednji zgled uredi in prešteje knjige o visual basicu po letih, vendar v izid vključi le tiste vrstice, v katerih je skupno število knjig večje od 5:

```
SELECT [Year Published] AS Leto, Count(*) AS [Izdanih knjig] FROM Titles
  WHERE Title LIKE '*visual basic*'
  GROUP BY [Year Published]
  HAVING Count(*) > 5
  ORDER BY [Year Published] DESC
```

Izbor iz več tabel

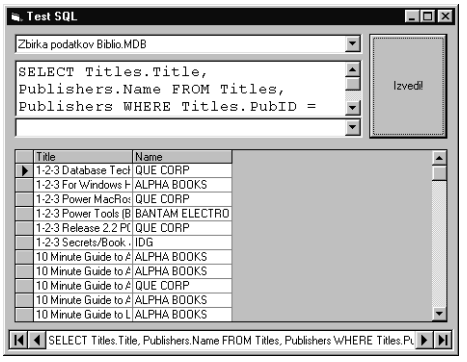
Stavek SELECT vedno vsebuje določilo FROM, v katerem v poljubnem vrstnem redu naštejemo tabele, iz katerih črpamo našteta polja. Doslej smo vedno uporabili eno samo tabelo, prava moč jezika SQL pa se pokaže pri bolj zapletenih poizvedbah, ki združujejo vsebino več tabel (*table joins*). To je tudi smisel uporabe relacijskega modela zbirke podatkov.

Če pri izboru zapisov iz več tabel naletimo na enako poimenovana polja, lahko imenski spor razrešimo z rabo operatorja sklicevanja, pike (.). Tudi če so imena polj popolnoma različna, je zapis imena skupaj s tabelo priporočljiv zaradi boljše berljivosti stavka.

Naslednji zgled izpiše naslov in založnika knjige. Vsebino polja PubID pojasni tako, da najde ime ustreznega založnika v njihovi tabeli:

```
SELECT Titles.Title, Publishers.Name
  FROM Titles, Publishers
  WHERE Titles.PubID = Publishers.PubID
  ORDER BY Titles.Title
```

Slika 13-7: Izbor zapisov iz dveh tabel



Če ne bi predpisali pogoja v določilu WHERE, bi dobili vse mogoče kombinacije polj iz dveh tabel. To je tako zelo veliko zapisov, da lahko njihovo obdelovanje za precej časa zamori še tako zmogljiv računalnik. S pogojem smo podatkovnemu stroju naložili, naj vrne le tiste zapise, pri katerih se oznaki založbe ujemata. Čar uporabe jezika SQL je v tem, da zgolj opišemo, kaj želimo kot izid. Umazane podrobnosti so stvar podatkovnega stroja.

Zunanje tabele

Narečje jezika SQL, ki ga uporablja podatkovni stroj Jet, omogoča vključevanje zunanjih tabel v poizvedbo. V določilu FROM lahko s predlogom IN določimo vrsto in pot do zunanje zbirke podatkov. Naslednja poizvedba uporabi zapise iz tabele »Tabela«, ki je zapisana v obliki dBASE IV v datoteki `d:\zbirke\dbase\podatki\Tabela.dbf`:

```
SELECT * FROM Tabela IN 'd:\zbirke\dbase\podatki' 'dBASE IV;'
```

Določilu IN podamo dva niza znakov, pot do zbirke in njeno vrsto. Opis različnih oblik zapisa zbirke podatkov smo spoznali, ko smo obravnavali lastnosti Connect gradnika Data. Čeprav je uporaba zunanjih zbirk neposredno v poizvedbi včasih priročna bližnjica, je uporaba priključenih tabel prek predmetov DAO precej bolj učinkovita.

Prepis izida v tabelo

Vsako izbirno poizvedbo SELECT lahko preobrazimo v posebno obliko akcijske poizvedbe, ki vrnjene zapise shrani v novo tabelo zbirke. Dovolj je, da ukazu SELECT dodamo še določilo INTO, ki ga spremlja ime nove tabele. Oglejmo si popravljeno različico zadnjega zgleda, ki bo izid poizvedbe prepisala v novo tabelo v domorodni zbirki podatkov:

```
SELECT * INTO PrepisanaTabela
  FROM Tabela IN 'd:\zbirke\dbase\podatki' 'dBASE IV;'
```

Tabela, v katero dodamo nove zapise, ne sme obstajati že od prej, drugače pride do izvajalne napake. Prepis izida poizvedbe je zelo priročen način za vmesno shranjevanje podatkov, izdelovanje rezervnih kopij ali rednih poročil, ki jih lahko izdelujemo npr. vsak mesec s povsem enako, ustrezno zapisano poizvedbo (taka poizvedba mora uporabiti izraz z datumsko funkcijo). Polja, ki jih prepišemo v drugo tabelo, ohranijo svoj tip in vsebino, vse druge informacije (indeksi, relacije, določila itn.) pa se ne prenesejo.

Akcijske poizvedbe

Z akcijskimi poizvedbami lahko zelo elegantno in hitro opravimo masovna odstranjevanja, popravljanja in dodajanja zapisov.

Odstranjevanje zapisov

Eden ali več zapisov iz poljubnega števila tabel odstranimo z ukazom DELETE. Simbolični zapis popolne oblike stavka je:

```
DELETE [tabela1.*, tabela2.* [, …]] FROM tabela1 [, tabela2 [,…]]
  [WHERE pogoj]
```

Ukaz DELETE vedno odstrani izbrane zapise v celoti. Če želimo pobrisati vsebino kakega polja, moramo uporabiti akcijsko poizvedbo za posodabljanje vsebine. Če pogoja ne predpišemo, odstranimo vse zapise, ohranimo pa *strukturo* tabele. Če tudi te ne potrebujemo, tabelo odstranimo mnogo hitreje s stavkom DROP TABLE.

Vsebino prej ustvarjene tabele bi popolnoma odstranili takole:

```
DELETE FROM PrepisanaTabela
```

Kadar želimo odstraniti zapise iz več tabel, moramo prepisati še seznam polj, ki našteva po eno polje za vsako tabelo, našteto v določilu FROM. Pri zapisu uporabimo operator pike (.), ki natančno opredeljuje pripadnost polja ustrezni tabeli. Ime polja ni pomembno, najbolj smiselno je zapisati kar posebni simbol (zvezdico, *), saj ta najbolje ponazarja delovanje ukaza. Odstranjevanje zapisov iz več tabel hkrati ni nič bolj učinkovito, kakor če z več stavki odstranimo zapise iz vsake posamezne tabele posebej.

Posodabljanje vsebine

Vsebino posameznih polj v obstoječih zapisih lahko posodobimo z uporabo stavka UPDATE. Njegova popolna oblika v simboličnem zapisu je:

```
UPDATE tabela1 [, tabela2 [, …]]
  SET polje1 = vrednost1 [, polje2 = vrednost2 [, … ]]
  WHERE pogoj
```

Z enim stavkom je mogoče posodobiti polja v več tabelah, vendar – podobno kakor pri stavku DELETE – to ni nič hitreje, kakor če za posodabljanje vsake tabele zapišemo samostojen stavek. Najbolj dobrodošla lastnost stavka UPDATE je možnost izračuna nove vrednosti iz obstoječih zapisov. Denimo, da imamo tabelo izdelkov in bi radi vse izdelke, ki prihajajo iz uvoza, podražili za 30 %:

```
UPDATE Izdelki SET Cena = Cena * 1.3 WHERE Izvor <> 'Slovenija'
```

Obnavljanje je akcijska poizvedba, ki ne vrača zapisov. Če želimo preveriti učinek poizvedbe, lahko zapišemo podobno izbirno poizvedbo:

```
SELECT Cena * 1.3 AS NovaCena FROM Izdelki WHERE Izvor <> 'Slovenija'
```

Kadar želimo pobrisati vsebino določenega polja, ga lahko nastavimo na privzeto vrednost za določen tip. S tem ima polje še vedno določeno vsebino, kar je lahko vsiljeno s strukturo zbirke podatkov. Če je to dopuščeno, lahko posamezno polje postavimo v nedoločeno vrednost (*null value*) z rezervirano besedo jezika SQL:

```
UPDATE Izdelki SET Cena = NULL
  WHERE Izvor IN ('Gornja Volta', 'Rodezija', 'Zaire')
```

Dodajanje zapisov

S stavkom INSERT INTO lahko v tabelo dodamo nov zapis ali več zapisov, ki so izid izbirne poizvedbe. Stavek zato lahko zapišemo v dveh oblikah. Dodajanje posameznega zapisa v simboličnem zapisu se glasi:

```
INSERT INTO tabela (polje1 [, polje2 [, …]])
  VALUES (vrednost1 [, vrednost2 [, …]])
```

Dodajanje več zapisov pa je kombinacija ukazov INSERT INTO in SELECT:

```
INSERT INTO tabela [ IN zunanjaZbirka ]
  SELECT polje1 [, polje2 [, …]])
  FROM iTabela1 [, iTabela2 [, …]]
  [ WHERE pogoj ]
```

Pri dodajanju posameznega zapisa moramo za vsako navedeno polje podati tudi njegovo vrednost. Če je v strukturi tabele določenih še več polj, dobijo ta nedoločeno vrednost (*null*), razen če tega ne preprečuje katera od omejitev za vzdrževanje celovitosti zapisa. Potem se sproži izvajalna napaka. Seznam polj lahko tudi izpustimo, če predpišemo vrednost za *vsako* polje v tabeli.

Pri množičnem dodajanju zapisov je koristno najprej preveriti samo izbirno poizvedbo in tako preveriti vsebino, ki se bo dodala v tabelo. Ukaz INSERT INTO vedno dodaja zapise na konec obstoječe tabele. Za prepis izida izbirne poizvedbe v novo tabelo uporabimo ukaz SELECT INTO (glej zgoraj).

Sestavljene poizvedbe

Sestavljene poizvedbe združujejo več stavkov SELECT v bolj zapletene izbirne poizvedbe ali kombinirajo ukaze akcijskih in izbirne poizvedbe. Združevanje posameznih podpoizvedb je mogoče z uporabo posebnih določil jezika SQL.

Omejitve podpoizvedb

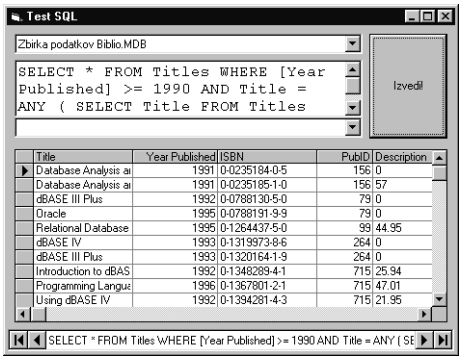
Izbirno podpoizvedbo priključimo obstoječi v izrazu znotraj določila WHERE ali HAVING. Za združitev več podpoizvedb imamo na voljo tri različne oblike zapisa:

- `primerjava [ { { ANY | SOME } | ALL } ] ( izbirnaPodpoizvedba )`
- `izraz [ NOT ] IN ( izbirnaPodpoizvedba )`
- `[ NOT ] EXISTS ( izbirnaPodpoizvedba )`

Če želimo izbrati zapise, ki se morajo po nekem merilu primerjanja ujemati z *vsaj* enim zapisom, ki ga vrne podpoizvedba, uporabimo predlog ANY ali SOME (njun pomen je enak). Naslednji zgled vrne vse knjige, izdane v 90. letih, katerih naslov je enak kateri izmed knjig, izdanih v prejšnjih desetletjih:

```
SELECT * FROM Titles
  WHERE [Year Published] >= 1990 AND
    Title = ANY ( SELECT Title FROM Titles WHERE [Year Published] < 1990 )
```

Slika 13-8: Izid sestavljene poizvedbe



Predlog ALL je mnogo strožji. Vrnjeni zapisi se morajo po merilu primerjanja ujemati z *vsemi* zapisi, ki jih vrne podpoizvedba. V zgornjem zgledu bi sprememba predloga iz ANY v ALL ne imela smisla.

S predlogom IN je mogoče nadomestiti kombinacijo »= ANY«. Zgornjo poizvedbo bi lahko zapisali tudi takole:

```
SELECT * FROM Titles
  WHERE [Year Published] >= 1990 AND
    Title IN ( SELECT Title FROM Titles WHERE [Year Published] < 1990 )
```

Lahko uporabimo tudi predlog NOT IN, enakovreden kombinaciji »<> ANY«. Potem bi dobili zapise, ki se ne ujemajo z nobenim iz podpoizvedbe.

Predlog EXISTS določa samostojen izraz, ki vrne resnico ali neresnico, odvisno od tega, ali v podpoizvedbi sploh je kak zapis ali ne.

Če v izrazu znotraj podpoizvedbe uporabljamo enako ime tabele kakor v osnovni poizvedbi ali če bi radi omejili izbor s poljem iz iste tabele v osnovni poizvedbi, lahko razrešimo imenski konflikt s preimenovanjem tabele z določilom AS, ki ga že poznamo iz uporabe pri izbiri polj. Naslednji zgled uporablja zbirko podatkov Nwind.MDB. Izberemo zaposlene, katerih starost v letih je nadpovprečna za njihovo delovno mesto:

```
SELECT * FROM Employees AS Emp1
  WHERE Year(BirthDate) < (
    SELECT AVG(Year(BirthDate))
      FROM Employees
      WHERE Emp1.Title = Employees.Title
  )
```

Povprečno starost izračunamo z vgrajeno funkcijo visual basica Year() in združevalno funkcijo jezika SQL. Opis delovnega mesta je v tabeli uslužbencev podan v polju »Title«. Ker moramo omejiti podpoizvedbo glede na opis delovnega

mesta osnovne poizvedbe, smo morali poseči po *sinonimu* tabele (*table alias*), ki ga vpeljemo z določilom AS.

Združevanje tabel

Že ko smo obravnavali izbirne poizvedbe, smo se srečali z združitvijo dveh tabel na podlagi posebej zapisanega merila. To merilo je iz vseh mogočih kombinacij izluščilo le tiste pare, ki so se ujemali v zapisanem izrazu.

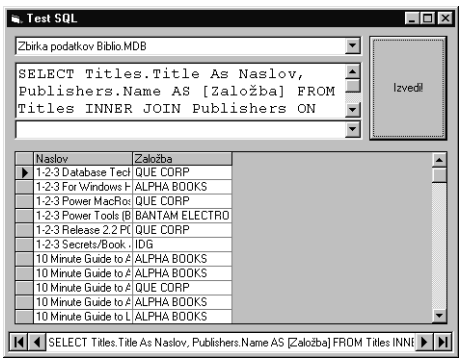
V jeziku SQL je mogoče dve tabeli relacijsko povezati tudi s posebnim operatorjem. Taka povezava združi (*join*) dve tabeli v novo, navidezno tabelo, ki jo lahko v nadaljevanju stavka uporabljamo povsem enakovredno z obstoječimi tabelami. Združitev tabel dosežemo na podlagi združljivega polja, ki si ga delita obe tabeli – *ključa*. Ponavadi gre za primarni ključ ene in tuji ključ druge tabele.

Način združevanja tabel s ključem lahko predpišemo na tri različne načine. V narečju SQL, ki ga uporablja podatkovni stroj Jet, se ta združevanja zapišejo znotraj določila FROM na tri različne načine:

- INNER JOIN. V novo tabelo so vključeni le tisti zapisi, pri katerih ima vsak ključ prve tabele ustrezen par v drugi (*notranje* združevanje, *inner-* ali *equi-join*).
- LEFT JOIN. V novo tabelo so vključeni vsi zapisi prve tabele in samo tisti zapisi iz druge, ki se ujemajo v ključu (*levo zunanje* združevanje, *left outer join*).
- RIGHT JOIN. V novo tabelo so vključeni vsi zapisi druge tabele in samo tisti zapisi iz prve, ki se ujemajo v ključu (*desno zunanje* združevanje, *right outer join*).

V zbirki knjig Biblio.MDB imamo različne tabele naslovov in njihovih založnikov. Če želimo izbrati podrobnejšo informacijo o posamezni knjigi, bomo uporabili združevanje tabel. Z notranjim združevanjem po ključu PubID lahko vsaki knjigi priredimo opisno ime založbe:

```
SELECT Titles.Title As Naslov, Publishers.Name AS [Založba]
  FROM Titles INNER JOIN Publishers ON Titles.PubID = Publishers.PubID
  ORDER BY Titles.Title
```



Slika 13-9: Izbirna poizvedba iz združenih tabel

NAMIG Uporabi šumnikov se pri poimenovanju elementov strukture zbirke podatkov izogibamo že ves čas. Pri določanju sinonimov polj pa bi bila njihova uporaba dobrodošla, saj npr. v gradniku Data Bound Grid uporabniku dodatno pojasnijo ime stolpca. Ker se nekateri znaki neangleških abeced znotraj podatkovnega stroja obravnavajo kot ločilo in ne kot črka, je najbolj varno, da s šumniki izraženo ime zapišemo v oglatih oklepajih ([]), tako kakor v zadnjem zgledu.

V izidu poizvedbe so zbrane samo tiste knjige, za katere je določen ključ založbe, ki mora biti še v drugi tabeli. Če bi želeli izbrati vse knjige, tudi tiste z neustrezno določeno založbo, uporabimo *levo* zunanje združevanje:

```
SELECT Titles.Title As Naslov, Publishers.Name AS [Založba]
  FROM Titles LEFT JOIN Publishers ON Titles.PubID = Publishers.PubID
  ORDER BY Titles.Title
```

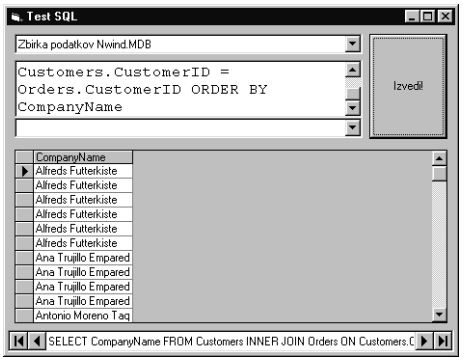
Če pa bi želeli izbrati vse opise založb, tudi tiste, ki jim ne pripada nobena knjiga, bi uporabili *desno* zunanje združevanje.

```
SELECT Titles.Title As Naslov, Publishers.Name AS [Založba]
  FROM Titles RIGHT JOIN Publishers ON Titles.PubID = Publishers.PubID
  ORDER BY Titles.Title
```

Naslednji zgled uporablja zbirko Nwind.MDB. Izbirna poizvedba vrne samo imena tistih podjetij, ki so vložila naročila:

```
SELECT CompanyName
  FROM Customers INNER JOIN Orders ON
    Customers.CustomerID = Orders.CustomerID ORDER BY CompanyName
```

Slika 13-10: Podvojeni zapisi v združeni tabeli



Poizvedba deluje, vendar vrne podvojene zapise, saj je lahko isto podjetje vložilo več naročil. Če jih želimo izločiti, uporabimo določilo DISTINCTROW:

```
SELECT DISTINCTROW CompanyName
  FROM Customers INNER JOIN Orders ON
    Customers.CustomerID = Orders.CustomerID ORDER BY CompanyName
```

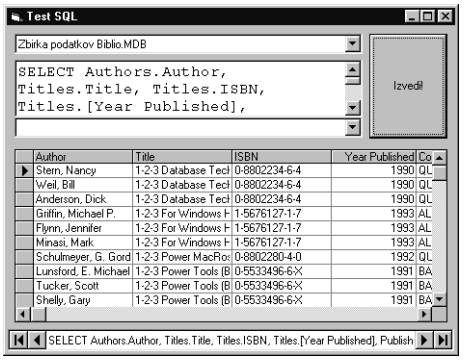
Pri izbiranju zapisov smo že spoznali določilo DISTINCT. Razlika med DISTINCT in DISTINCTROW je v tem, da se slednji nanaša na *celoten* zapis (vrstico) v navidezni, združeni tabeli, ki nastane kot posledica združevanja. Zato se to določilo pri uporabi ene same tabele zanemari. Tam nam pride prav prvo določilo DISTINCT, ki smo ga že spoznali pri enostavnih izbirnih poizvedbah.

> **NAMIG** Če bi pri zadnji poizvedbi namesto DISTINCTROW uporabili določilo DISTINCT, bi dobili enak izid, saj s tem najprej izberemo enolična imena podjetij iz tabele strank, ki ga posredujemo v združevanje. Pri poizvedbi z združevanjem, ki izbere več polj iz različnih tabel, moramo uporabiti določilo DISTINCTROW, sicer se lahko zgodi, da vrnjeni izid ne bo pravilen.

Združevanja lahko gnezdimo in s tem ustvarimo poizvedbo, ki združi tri ali več tabel. Naslednja poizvedba uporabi vse štiri tabele iz zbirke Biblio.MDB, da vrne vse opisne vrednosti vseh podatkov, ki pripadajo posameznemu naslovu:

```
SELECT Authors.Author, Titles.Title, Titles.ISBN, Titles.[Year Published],
Publishers.[Company Name]
  FROM
   ( ( [Title Author] INNER JOIN Titles
         ON [Title Author].ISBN =  Titles.ISBN )
     INNER JOIN Authors
       ON [Title Author].Au_ID = Authors.Au_ID )
   INNER JOIN Publishers
     ON Titles.PubID = Publishers.PubID
  ORDER BY Titles.Title;
```

Slika 13-11: Združena poizvedba štirih tabel



Najprej smo z dvema združitvama razrešili povezavo »mnogi z mnogimi«, pri čemer nam pomaga vmesna tabela »Title Author«. To najprej povežemo s tabelo naslovov, dobljeno navidezno tabelo pa z avtorji. Na koncu razrešimo še šifre založnikov. Pri gnezdenju združevanj moramo uporabiti oklepaje, s katerimi omejimo posamezne podzdružitve.

Združevanje poizvedb

Več neodvisnih poizvedb je mogoče združiti v unijo. Vse poizvedbe morajo vračati enako število polj, ni pa pomembno, kakšen je njihov tip ali dolžina. Polja dobijo ime po prvi poizvedbi. Če želimo, lahko celotno unijo zapisov uredimo po merilu, ki ga zapišemo z določilom ORDER BY čisto na koncu.

V simbolični obliki se glasi zapis unije poizvedb takole:

```
poizvedba1 UNION [ALL] poizvedba2 [ UNION [ALL] poizvedba3 [ … ] ]
  [ ORDER BY kriterij ]
```

Po privzeti nastavitvi unija ne vrača podvojenih zapisov. Z določilom ALL lahko zahtevamo vključitev vseh zapisov v unijo, kar tudi pospeši izvajanje poizvedbe.

Značilnosti Jet SQL

Ko smo obravnavali jezik SQL, smo poudarjali značilnosti, ki niso značilne za vse izvedbe jezika SQL, ampak so posebnost narečja, ki ga uporablja podatkovni stroj Jet. Nekaterih še nismo posebej opisali, zato se bomo sedaj še enkrat ozrli po njih. Najprej pa si oglejmo še eno značilnost.

Optimalizacija poizvedb

Priljubljenost jezika SQL temelji na njegovi preprostosti. Vse operacije, od ustvarjanja do učinkovite rabe zbirke podatkov, je mogoče opisati z nekaj relativno enostavnimi stavki v jeziku, ki spominja na angleščino. To je mogoče zaradi osnovne značilnosti jezika:

Stavki v jeziku SQL ne določajo natančno poteka operacije, ampak samo njen izid.

Ker je večina dela prepuščena podatkovnemu stroju ali strežniku, je dostikrat učinkovitost poizvedbe neposredno odvisna od njegove prilagodljivosti in sposobnosti *optimalizacije* poizvedb. Velika večina programske opreme, ki dopušča uporabo jezika SQL, vsebuje posebne optimalizacijske postopke, ki poizvedbo prevedejo v kar najučinkovitejše delo s podatki. Take postopke pozna tudi podatkovni stroj Jet. V tržne namene so jih pri Microsoftu poimenovali »tehnologija Rushmore«. Čeprav se optimalizacijski postopki sami potrudijo pri vsaki poizvedbi, jih z nekaj namigi v ukazih jezika SQL lahko bolje izkoristimo. Pregled nekaterih zmožnosti tehnologije Rushmore opisuje spodnja tabela:

Zmožnost optimalizacije	Opis
Preprosti izrazi	Preprost izraz je sestavljen iz primerjave med indeksiranim poljem in poljubno vsebino. Če za primerjavo uporabimo operator LIKE, moramo vzorec primerjanja zapisati z znanim začetkom. Optimalizacija pri vzorcih vrste `'*podniz*'` ni mogoča.

Zmožnost optimalizacije	Opis
Sestavljeni izrazi	Sestavljeni izraz, ki ga je mogoče optimalizirati, je iz dveh ali več preprostih izrazov, združenih z logičnima operatorjema AND ali OR.
Uporaba indeksov	Če indeks vključuje več polj, je optimalizacija mogoča le v izrazih, ki uporabijo začetna polja. Npr. za indeks, ki vključuje priimek in ime, je optimalizacija mogoča pri uporabi priimka ali pri uporabi priimka in imena, ni pa mogoča pri uporabi imena. Pri indeksih, ki urejajo podatke padajoče, je optimalizacija mogoča le s primerjalnim operatorjem enakosti (=).
Združevalne funkcije	Optimalizacija je mogoča za preštevanje zapisov, zapisano v obliki COUNT(*).
Združevanje poizvedb	Hitrejše izvajanje združenih poizvedb dosežemo z rabo določila UNION ALL.
Oblike zbirk	Tehnologija Rushmore omogoča optimalizacijo poizvedb, ki uporabljajo domorodne zbirke ali zbirke v zapisih dBase in Fox Pro. Pri drugih virih, vključno z viri ODBC, optimalizacija ni mogoča[55].

Za kar najučinkovitejše izvajanje poizvedb je pomembna že sama struktura zbirke podatkov, predvsem pravilno določeni in dovolj številni indeksi.

Posebnosti narečja

Jezik SQL je standardiziran z več določili ameriške ustanove ANSI (*American National Standard Institute*). Narečje, ki ga uporablja podatkovni stroj Jet (Jet SQL), je večinoma skladen z določili standarda SQL ANSI-89 Level 1. Nekaj elementov tega standarda je izpuščenih, nekatere svojevrstne rešitve pa so dodane. Teh razlik se je treba zavedati, če v prihodnosti načrtujemo prenos zbirke v podatkovni strežnik. Poglavitne razlike so:

- Jet SQL pozna več razpoložljivih podatkovnih tipov za polja.
- Jet SQL pozna nova ukaza. TRANSFORM omogoča združevanje po poljih v izbirni poizvedbi, PARAMETERS pa omogoča napoved parametrov poizvedbe. Ukaz SELECT je dopolnjen z določilom WITH OWNER ACCESS, ki omogoča uporabo in vpogled v začitene zapise z vnaprej pripravljenimi poizvedbami.
- Jet SQL pozna nov predlog TOP za izločanje nekaj prvih zapisov in določilo IN za neposredni priklop tabel iz zunanje zbirke podatkov.

[55] Pri virih ODBC se poizvedba posreduje naprej. Zmogljivi podatkovni strežniki vsi po vrsti vključujejo optimalizacijske postopke, ki lahko znatno prekašajo tehnologijo Rushmore.

- Jet SQL pozna novi, statistični združevalni funkciji: StDev (standardno odstopanje) in VarP (variacija). V izrazih lahko uporabljamo tudi vgrajene funkcije visual basica.
- Drugačna uporaba primerjalnih operatorjev BETWEEN in LIKE. V ANSI SQL morata biti meji pri operatorju BETWEEN našteti po naraščajoči vrednosti, pri Jet SQL vrstni red ni pomemben. Za zapis vzorca primerjave uporablja ANSI SQL znaka podčrtaj (_) za poljuben znak in odstotek (%) za poljubno zaporedje, Jet SQL pa znaka vprašaj (?) in zvezdica (*).
- Jet SQL vsiljuje manj omejitev pri združevanju in urejanju zapisov. Urejati je namreč mogoče tudi po vrednosti izrazov.

Natančen opis naštetih zmožnosti najdemo v sprotni pomoči. Uporabo parametriziranih poizvedb si bomo natančneje ogledali tudi v nadaljevanju.

Visual basic in SQL

Posamezni predmeti podatkovnega modela DAO so posebej uporabni prav skupaj s stavki v jeziku SQL. Nekatere smo spoznali že v prejšnjem poglavju, nekatere pa smo prihranili za nadaljevanje, saj jih je mogoče učinkovito izkoristiti samo, če dobro poznamo jezik SQL.

Posredovanje ukazov

Kakor smo že omenili, je jezik SQL ločen od visual basica. Ukaze zapišemo v obliki niza znakov, ki ga uporabimo za nastavitev lastnosti ali kot parameter pri klicu metode. V zgledu, ki nam je v tem poglavju rabil za ponazoritev ukazov v jeziku SQL, smo že uporabili obe možnosti:

- Izbirne poizvedbe lahko uporabimo pri določitvi lastnosti DataSource gradnika Data ali pri ustvarjanju zbirke zapisov z ukazov OpenRecordset.
- Vse druge poizvedbe uporabimo neposredno v zbirki podatkov z metodo **Execute**.

Ker je vsak ukaz v jeziku SQL znotraj visual basica samo niz znakov, ga lahko pred uporabo preoblikujemo ali mu dodamo spremenljive dele. Če npr. želimo, da uporabnik z vnosom določi, katere naslove iz zbirke knjig bi želel videti, lahko zapišemo naslednji izraz:

```
strSQL = SELECT * FROM Titles WHERE Title > '" & Text1.Text & "'"
```

ki ga posredujemo naprej metodi OpenRecordset:

```
Set rs = OpenRecordset(strSQL)
```

Ne pozabimo, da moramo besedilo omejiti z narekovaji, datumske konstante pa z znakoma *hash* (#).

Pripravljene poizvedbe

Znotraj domorodne oblike zapisa zbirke podatkov (Access, MDB) je mogoče shraniti tudi pripravljene poizvedbe, ki jih po potrebi lahko parametriziramo. Poizvedbe določa zbirka **QueryDefs**. Spodnji zgled izpiše že določene poizvedbe, potem pa določi novo:

```
Dim db As Database
Dim qd As QueryDef

  Set db = OpenDatabase("Biblio.MDB")

  ' Obstoječe pripravljene poizvedbe
  For Each qd In db.QueryDefs
    MsgBox qd.Name & vbCrLf & vbCrLf & qd.SQL
  Next

  ' Nova poizvedba
  Set qd = db.CreateQueryDef("Naslovi letos")
  qd.SQL = "SELECT * FROM Titles WHERE [Year Published] = Year(Now)"

  db.Close
```

Predmeti razreda **QueryDef** se vedejo drugače kakor drugi predmeti, ki sestavljajo strukturo zbirke podatkov. S klicem metode **CreateQueryDef** se poimenovana poizvedba že trajno doda v samo definicijo zbirke podatkov. Metoda Append zato ni dovoljena. Shranjene poizvedbe lahko odstranimo enako kakor pri drugih zbirkah, z metodo Delete.

Če želimo samo začasen predmet razreda QueryDef, ga moramo ustvariti brez imena (nujno moramo posredovati prazen niz):

```
Set zacasniQd = db.CreateQueryDef("")
```

V predmet QueryDef lahko shranimo poljuben ukaz v jeziku SQL. Poseganje v strukturo zbirke in izvajanje akcijskih poizvedb dosežemo tako, da nad predmetom izvedemo metodo **Execute**. Naslednji zgled ustvari začasno akcijsko poizvedbo in jo uporabi:

```
Set qd = db.CreateQueryDef("", "CREATE TABLE Tabela (f1 TEXT (50), f2 LONG)")
qd.Execute
```

Uporabo izbirnih poizvedb si bomo ogledali v nadaljevanju.

Parametrizacija

Pripravljene poizvedbe v predmetih razreda QueryDef so še posebej primerne v parametrizirani obliki. Taka poizvedba je določena splošno, pred njeno rabo pa moramo poskrbeti še za vzpostavitev parametrov. Oglejmo si zgled, ki bo uporabil začasen izvod razreda QueryDef:

```
Dim db As Database
Dim strSQL As String
Dim qd As QueryDef
Dim rs As Recordset

  Set db = OpenDatabase("Biblio.MDB")

  ' Parametrizirana poizvedba
  ' - niz s parametrizirano poizvedbo
  strSQL = "PARAMETERS [Vzorec naslova] TEXT, [Letnica] DATE; " & _
    "SELECT * FROM Titles WHERE " & _
      "[Year Published] = [Letnica] AND Title LIKE [Vzorec naslova]"
  ' - stvaritev začasnega predmeta brez imena
  Set qd = db.CreateQueryDef("", strSQL)
  ' - določanje parametrov
  qd.Parameters("[Vzorec naslova]") = "M*"
  qd.Parameters("[Letnica]") = 1995
  ' - uporaba poizvedbe
  Set rs = qd.OpenRecordset(dbOpenForwardOnly)
  ' - izpis izida
  Cls
  Do Until rs.EOF
    Print rs!Title
    rs.MoveNext
  Loop

  db.Close
```

Jet SQL pozna poseben ukaz PARAMETERS, s katerim lahko določimo parametre poizvedbe. Naštejemo jih na seznamu, skupaj z njihovim tipom. Ker ukazu pripada poseben stavek v SQL, ga od izbirne poizvedbe ločimo s podpičjem (;), ki je ločilo stavkov. Tako določene parametre lahko uporabimo znotraj izrazov določil WHERE in HAVING. Vsi določeni parametri se v predmetu razreda QueryDef odrazijo v zbirki **Parameters**. Pred uporabo same poizvedbe jim priredimo končno vrednost. Opazimo lahko, da so posamezni parametri različnih podatkovnih tipov, točno tako, kakor so bili določeni v stavku SQL.

Pri ustvarjanju parametriziranih poizvedb moramo posredovati niz z ukazom v SQL neposredno metodi CreateQueryDef. Naknadno določanje ukaza z lastnostmi SQL bo pri uporabi poizvedbe povzročilo napako.

Imena parametrov je vedno dobro zapisati med oglatima oklepajema. Nekatera orodja, ki tudi uporabljajo Jet SQL (priloženi Visual Data Manager, Microsoft Access itn.) znajo pri uporabi parametriziranih poizvedb s tako označenimi imeni parametrov samodejno prikazati pogovorno okno za vnos vrednosti parametra. Tako vedenje dosežemo tudi v visual basicu s programsko kodo v prestrezovalniku napake.

```
' Poizkus uporabe parametrizirane poizvedbe
On Error GoTo Napaka
  Set rs = qd.OpenRecordset(dbOpenForwardOnly)
On Error GoTo 0
```

```
'  . . .

  Exit Sub

Napaka:
Dim par As Parameter
Dim strPar As String

  ' Manjkajoči parametri?
  If Err.Number = 3061 Then
    For Each par In qd.Parameters
      strPar = InputBox("Vnesi vrednost za " & par.Name, _
        "Podaj parameter", par.Value)
      Select Case par.Type
        Case dbText
          par.Value = strPar
        Case Else
          par.Value = CDbl(strPar)
      End Select
    Next
    Resume
  End If

  Resume Next

End Sub
```

NAMIG Podobno je zapisan tudi prestrezovalnik napake v orodju Visual Data Manager, le da tisti ne preverja podatkovnega tipa parametrov, zato se pri netekstovnih parametrih ne obnese dobro.

Varnost

V varnih delovnih prostorih, ki jih vzpostavljamo s sistemsko zbirko podatkov o delovnih skupinah (varne delovne prostore smo natančneje obravnavali na koncu prejšnjega poglavja), je mogoče pripraviti poizvedbe, ki so na voljo vsem uporabnikom, ne glede na njihove dostopne pravice. Lastnik določenega elementa lahko pripravi *javno* shranjeno poizvedbo s posebnim določilom jezika SQL, ki dopušča njegovo uporabo vsem.

Oglejmo si zgled take poizvedbe:

```
Dim strSQL As String
Dim qd As QueryDef

  ' ...

  ' Javna poizvedba
  strSQL = "SELECT Ime, Priimek FROM Zaposleni " & _
             "ORDER BY Priimek, Ime " & _
             "WITH OWNERACCESS OPTION"

  Set qd = db.CreateQueryDef("Pregled zaposlenih", strSQL)
```

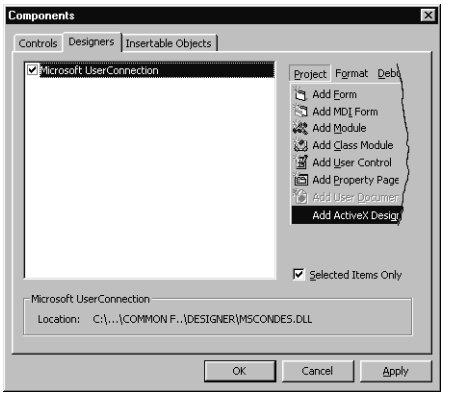
Določilo WITH OWNERACCESS OPTION določa, da se bo poizvedba »Pregled zaposlenih« vedno izvajala s pravicami njenega stvaritelja. S tem določilom lahko zapišemo vsako obliko poizvedbe, npr. tudi akcijsko, ki odstrani zapise v sicer nedostopnih tabelah.

Izgradnja poizvedb

Najboljša različica visual basica, Enterprise Edition, pozna poseben dodatek, ki omogoča vizualno izgradnjo poizvedb in njihovo uporabo v programski kodi. Taka orodja se imenujejo *načrtovalci* (*designers*). Z enim izmed njih se ukvarjamo že od samega začetka, to je v visual basic vgrajeni Form Designer, s katerim načrtujemo uporabniški vmesnik. Naprodaj je še več načrtovalcev, ki jih lahko naknadno vključimo v razvojno okolje visual basica.

Dodatne načrtovalce, ki so nam na voljo, lahko dodamo v projekt z izbiro ukaza **Components**... iz glavnega menuja **Project**. Sedaj moramo izbrati jeziček **Designers**, da prikažemo seznam razpoložljivih načrtovalcev. Želeni načrtovalec omogočimo z vklopom ustreznega stikala.

Slika 13-12: Dodajanje načrtovalcev



Odslej je v menuju **Project** na voljo podmenu **Add ActiveX Designer**, prek katerega omogočimo rabo vključenega načrtovalca (tako kakor nakazuje pogovorno okno na sliki zgoraj).

Načrtovalec poizvedb je še posebej uporaben pri povezovanju z zmogljivimi podatkovnimi strežniki. Njihovi rabi je posvečeno naslednje poglavje.

14. POGLAVJE

Strežnik in odjemalec

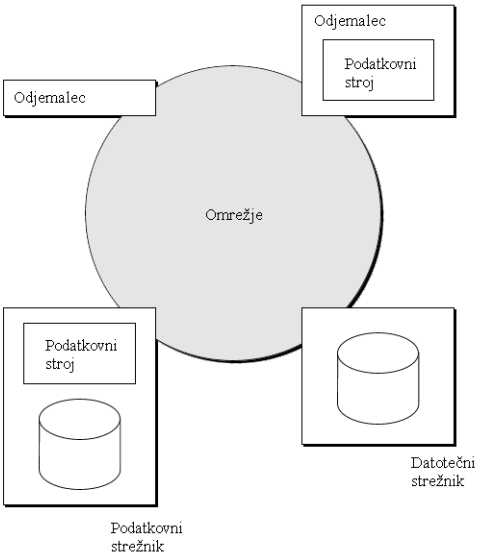
Za manjše in srednje velike programe, ki uporabljajo zbirke podatkov, je model zbirke v datoteki in podatkovnega stroja, ki je sestavni del programa za njeno uporabo, večinoma ustrezen. Tovrstni model uporabljajo tako imenovane »namizne« zbirke podatkov (*desktop databases*). Značilni predstavniki namiznih zbirk so programi, kot so Microsoft Access, Corel Paradox ali Borland dBase. Namizne uporabniške zbirke večinoma podpirajo hkratno delo več uporabnikov, podatki pa so lahko zbrani v *datotečnem* strežniku (*file server*) nekje v omrežju. Vsi zgledi, ki smo jih razvili v prejšnjih poglavjih, so uporabljali namizni model.

Zmogljivejši programi z veliko uporabniki in temu primernim številom sočasnih dostopov do podatkov zahtevajo pristop, ki presega zmožnosti namiznega modela. Tovrsten pristop temelji na zmogljivem računalniku, v katerega je nameščena namenska programska oprema za upravljanje zbirk podatkov – *podatkovnem strežniku*. Značilni predstavniki podatkovnih strežnikov (*database servers*) so Microsoft SQL Server, Oracle ali Informix. Storitve strežnika uporabljajo odjemalci, razpršeni po omrežju, zato se ta model imenuje *odjemalec/strežnik*. Prednosti modela odjemalcev in strežnika (*client/server*) so:

- Zanesljivejše in bolj trdoživo delovanje, ki temelji na enem samem, visoko zmogljivem podatkovnem strežniku.
- Posebne zmogljivosti, kot so dnevnik transakcij, zaščitno kopiranje ali obnavljanje ob izpadih.
- Enostavnejša širitev zmogljivosti (*scalability*) v primerjavi z namiznimi zbirkami.
- Zmogljivost odjemalcev praktično ne vpliva na učinkovitost dela.
- Manjša obremenjenost omrežja.

Slika 14-1 prikazuje delovanje obeh modelov dostopa do podatkov.

Pri uporabi datotečnega strežnika mora imeti podatkovni stroj vsak odjemalec, podatki pa so v datotečnem strežniku. Vse zahteve se obdelajo pri odjemalcu, strežnik rabi zgolj za shranjevanje datoteke, iz katere podatkovni stroj črpa



Slika 14-1: Dva modela dostopa do podatkov

zahtevane podatke. Pri podatkovnem strežniku sta skupaj zbrana podatkovni stroj in podatki. Odjemalec posreduje svoje zahteve strežniku kot visoko nivojske ukaze v jeziku SQL in prejme samo izid poizvedbe.

> **NAMIG** Delno lahko vedenje virov SQL posnemamo z uporabo gonilnikov za namizne zbirke podatkov (Access, dBase itn.). Seveda pa je obravnava v nadaljevanju namenjena predvsem tistim bralcem, ki lahko uporabljate storitve podatkovnega strežnika. Ta je lahko nameščen v vašem računalniku ali pa vam je dostopen kjerkoli v omrežju.

Način dostopa

V visual basicu imamo dostop do vseh virov ODBC, ki so v sistemu na voljo. Dostop je mogoč na dva načina:

- Prek podatkovnega stroja Jet.
- Mimo podatkovnega stroja Jet z uporabo neposrednega dostopa do virov ODBC.

Vsak pristop ima svoje prednosti in pomanjkljivosti, ki si jih bomo ogledali v nadaljevanju.

Uporaba podatkovnega stroja Jet

Posebne zmogljivosti pri dostopu do podatkovnih strežnikov, ki jih omogoča samo podatkovni stroj Jet, so:

- Zmožnost posodabljanja polj v združeni tabeli. Zbirko zapisov v obliki Recordset, ki je izid poizvedbe z združevanjem tabel (*table join*), je mogoče posodabljati kot katerokoli drugo tabelo.
- Vse poizvedbe se izvajajo znotraj implicitne transakcije, zato niso nikoli izvedene samo delno. Če se izvajanje poizvedbe vmes prekine, se obnovi prejšnje stanje.
- Uporaba zunanjih tabel. S predmeti DAO ali stavki v jeziku SQL se je mogoče navezati na tabelo v zunanji zbirki podatkov in jo uporabljati brez omejitev.
- Zaporedno iskanje po zbirki zapisov z metodami Find.
- Omogočene posebnosti Jet SQL, kot je npr. določanje parametričnih poizvedb.
- Poseganje v strukturo zbirke podatkov s predmeti modela DAO.
- Uporaba vezanih gradnikov.

Če naštetih prednosti ne potrebujemo, je večinoma učinkovitejša uporaba neposrednega dostopa do virov ODBC. Na srečo je mogoče oba načina dostopa kombinirati in izkoristiti njune prednosti.

Neposredni dostop

Neposredni dostop do virov ODBC (ODBC *Direct*) preusmerja dostop do nekaterih predmetov modela DAO programski knjižnici RDO (*Remote Data Objects*)[56]. Ta knjižnica popolnoma zaobide podatkovni stroj Jet, zato zahteva minimalen nabor sistemskih virov odjemalca, vendar ne pozna vseh njegovih zmogljivosti. Zahteve v jeziku SQL se posredujejo neposredno podatkovnemu strežniku. Prednosti neposrednega dostopa so:

- Z neposrednim dostopom je omogočeno maksimalno izkoriščanje prednosti podatkovnega strežnika.
- Manjša poraba sistemskih virov pri odjemalcu. Manjši promet v omrežju.
- Asinhrone poizvedbe. Med izvajanjem dolgotrajnejše poizvedbe je mogoče nadaljevati delo, kar omogoča razvoj učinkovitejših odjemalcev.
- Paketno posodabljanje vsebine. Več popravkov je mogoče shraniti na odjemalni strani in jih zatem v paketu poslati strežniku.

[56] Predmeti RDO so določeni v deljeni knjižnici DLL.

Gonilniki ODBC

Če želimo uporabljati podatkovne strežnike, jih moramo najprej napovedati v sistemu in zanj priskrbeti ustrezne gonilnike ODBC (*Open Database Connectivity*). Danes je standard ODBC široko sprejet, zato ustrezne gonilnike prejmemo praktično z vsakim podatkovnim strežnikom. Vsi gonilniki za isti podatkovni vir niso enako dobri, zato je najbolje, če imamo domorodne podatkovne gonilnike izdelovalca samega podatkovnega strežnika. Boljši različici visual basica vsebujeta gonilnike ODBC za podatkovna strežnika Microsoft SQL Server in Oracle.

NAMIG Če želimo namestiti priložene gonilnike ODBC, moramo izbrati polno namestitev visual basica professional oziroma enterprise. Gonilnike je mogoče namestiti tudi naknadno, na plošči CD-ROM visual basica je ustrezen namestitveni program v mapi \VB\ODBC\. Z nakupom visual basica smo dobili dovoljenje za razširjanje teh gonilnikov tudi v druge sisteme, ki jih želimo opremiti s programi za dostop do podatkovnih strežnikov.

Uporaba strežnika

V nadaljevanju si bomo podrobneje ogledali oba načina dostopa do virov ODBC. Preden začnemo te vire izkoriščati, jih moramo vzpostaviti v sistemu z njihovim gonilnikom, ki mora biti nameščen že prej (glej zadnji namig).

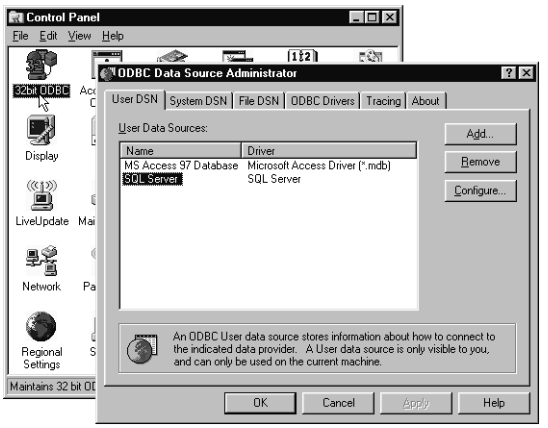
Vzpostavitev podatkovnega vira

V sistemskih nastavitvah najdemo pripomoček ODBC Data Source Administrator, ki se skriva pod ikono 32-bit ODBC (razvidno s slike 14-2). Z njim vzpostavimo ključe v sistemskem registru, ki so pomembni za vse programe, ki želijo izkoriščati vire ODBC.

Nove podatkovne vire lahko vzpostavimo ročno ali s programsko kodo. Postopek vzpostavitve je razmeroma enostaven, vendar se lahko od primera do primera razlikuje.

Ročna vzpostavitev

Pripomoček ODBC Data Source Administrator vključuje več nastavitev, ki se skrivajo pod različnimi jezički. Jeziček **ODBC Drivers** skriva seznam nameščenih gonilnikov za vire ODBC. Te vire lahko izpostavimo programom prek simboličnega imena, ki ga določa nastavitev DSN (*Data Source Name*). Na voljo so tri različne vrste



Slika 14-2: Upravljanje virov ODBC

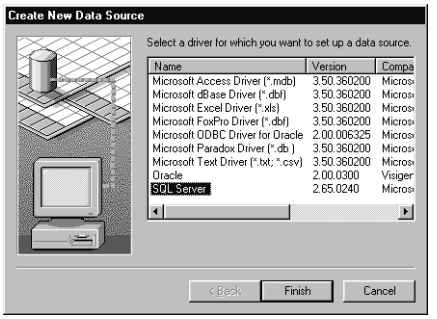
simboličnih imen DSN: User, System in File, vsaka s svojim jezičkom. Njihovo primerjavo najdete v spodnji tabeli:

Vrsta simboličnega imena vira	Opis
User DSN	Določa vire, ki so dostopni samo programom v istem računalniku.
System DSN	Določa vire, ki so dostopni v vsem omrežju, v katerega je vključen tudi naš računalnik.
File DSN	Določa vire, ki so predstavljeni z datotekami v posebni mapi. Te vire lahko souporabljajo uporabniki v omrežju, ki imajo nameščene enake gonilnike virov ODBC.

NAMIG Datoteke, ki ponazarjajo podatkovne vire vrste »File DSN«, so ponavadi shranjene v mapi C:\Program Files\Common Files\ODBC\Data Sources\. Po želji lahko pot do datotek tudi spremenimo. Pod jezičkom **File DSN** s seznamom Look spremenimo mapo in jo potrdimo za privzeto z gumbom Set Directory. Tako lahko izberemo tudi kako mapo v omrežju.

Ko izberemo želeno vrsto simboličnega imena (denimo User DSN), dodamo nov vir z gumbom **Add**... Vsak vir moramo najprej opredeliti, tako da s seznama razpoložljivih gonilnikov izberemo njegovega (npr. SQL Server). Pri virih, predstavljenih kot »File DSN«, sledi še nekaj korakov, sicer pa prvo fazo določanja gonilnika končamo s klikom gumba **Finish**.

Slika 14-3: Izbor gonilnika

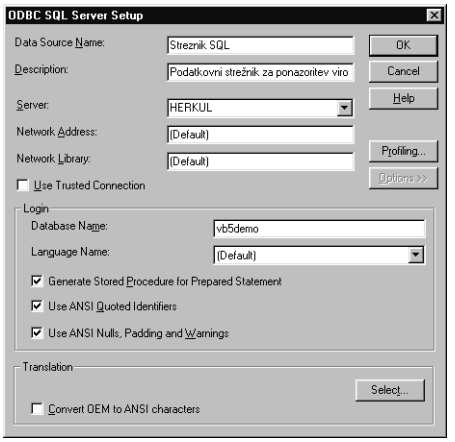


Nadaljnji postopek vzpostavitve je popolnoma odvisen od izbranega gonilnika. Pri vseh gonilnikih je treba izbrati simbolično ime vira in ime strežnika, druge nastavitve pa se lahko razlikujejo in niso vedno obvezne. Vse manjkajoče informacije lahko zahteva gonilnik ob poskusu vzpostavitve povezave iz programa.

Mi si bomo ogledali tipično vzpostavitev vira v strežniku SQL Server. Gonilnik prikaže pogovorno okno, ki zahteva naslednje nastavitve:

- V vnosno polje **Data Source Name** vnesemo poljubno simbolično ime vira, npr. »Streznik SQL«.
- V vnosno polje **Description** lahko vnesemo daljši opis vira.
- Na kombinirani seznam **Server** vnesemo *ime* strežnika (brez vodilnih »\\«) ali njegov internetni naslov.
- Ker podatkovni strežnik SQL Server lahko nadzoruje več zbirk podatkov, je mogoče nastaviti želeno zbirko, ki nas zanima. Ko kliknemo gumb **Options,** se prikažejo dodatne nastavitve. Ime želene zbirke podatkov vnesemo v polje **Database Name**.

Slika 14-4: Ročna vzpostavitev gonilnika



- Vnos končamo z gumbom OK. Na seznamu simboličnih imen virov sedaj najdemo novo vzpostavljeni vir. Nastavitve vira lahko spremenimo z gumbom **Configure**... Odstranimo pa ga z gumbom **Remove**.

Kakor smo že omenili, razen nastavitve simboličnega imena (Data Source Name) in imena strežnika nobena druga nastavitev ni obvezna. Za vse nas lahko povpraša gonilnik ob poizkusu navezave na vir.

> **NAMIG** Opis drugih nastavitev je na voljo v sprotni pomoči, dosegljivi s klikom gumba **Help**. V tem primeru se prikaže datoteka s pomočjo, ki spremlja izbrani gonilnik ODBC.

Vzpostavitev iz kode

Program v visual basicu bo bolj trdoživ, če za vzpostavitev vira poskrbimo iz programske kode. Pri tem nam pomaga metoda **RegisterDatabase** predmeta DBEngine, ki lahko vzpostavi ali posodobi uporabniški vir (User DSN). Oglejmo si zgled:

```
Dim strOpis As String

  strOpis = "Description=Podatkovni strežnik za ponazoritev virov ODBC"
  strOpis = strOpis & vbCrLf & "Server=localhost"
  strOpis = strOpis & vbCrLf & "Database=Pubs"
  strOpis = strOpis & vbCrLf & "OemToAnsi=No"

  DBEngine.RegisterDatabase "Streznik SQL", "SQL Server", True, strOpis
```

Metoda zahteva štiri parametre. Prvi je simbolično ime vira, drugi je ime izbranega gonilnika, tretji parameter določa, ali naj se ob manjkajočih informacijah oglasi gonilnik s svojim pogovornim oknom (*False*) ali ne (*True*). Zadnji parameter je niz z nastavitvami parametrov gonilnika. Nastavitve so med seboj ločene s prehodi v novo vrsto, kakor je razvidno iz zgleda. Če z metodo naslovimo ime obstoječega uporabniškega vira, se njegove nastavitve posodobijo v skladu s posredovanimi.

> **NAMIG** Nastavitve se shranijo v sistemski register pod ključ HKEY_LOCAL_MACHINE\Software\ODBC\Odbc.ini\. Tam lahko preberemo tudi imena drugih parametrov.

Do podatkov prek stroja Jet

Tudi pri uporabi virov ODBC prek podatkovnega stroja Jet lahko posežemo po predmetih DAO ali pa uporabimo gradnik Data in vezane gradnike. Prednost uporabe podatkovnega stroja Jet je prav v tem, da nam tudi oddaljene vire predstavi kar najbolj podobno domorodnim zbirkam podatkov.

Niz Connect

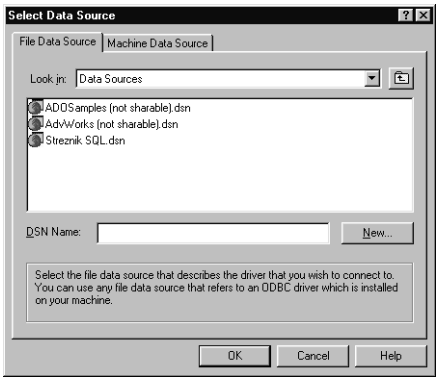
Če želimo dostop do virov ODBC, moramo pravilno določiti niz Connect, ki opisuje povezavo. Pregled vseh možnih nastavitev tega niza pri uporabi gonilnika strežnika SQL Server smo si ogledali že, ko smo obravnavali gradnik Data in njegove lastnosti Connect. Na splošno je niz sestavljen iz več nastavitev, ločenih s podpičjem (;). Prva nastavitev mora opredeljevati vrsto vira, druge pa si lahko sledijo v poljubnem zaporedju. Določila so lahko precej odvisna od gonilnika vira ODBC, na katerega se želimo priključiti.

Za najpreprostejšo določitev dostopa do vira ODBC zadošča, da niz nastavimo na vrednost:

```
ODBC;
```

V tem primeru bo visual basic za vse nastavitve zaprosil pripomoček ODBC Data Source Administrator, ki bo prikazal vrsto pogovornih oken.

Slika 14-5: Če ne določimo vira, nas pred vzpostavitvijo povezave pričaka pogovorno okno.



Če želimo uporabniku olajšati delo, določimo manjkajoče parametre sami. Zgled niza, ki natančneje opredeli dostop do vira, bi bil:

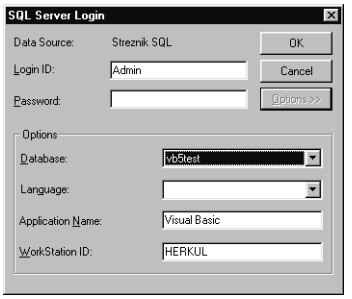
```
ODBC;DSN=Streznik SQL;
```

V tem primeru smo določili simbolično ime vira, ki ga želimo izkoriščati. Po morebitnih manjkajočih parametrih nas sedaj povpraša gonilnik, določen s simboličnim imenom. V našem primeru je to gonilnik strežnika SQL Server, ki pred vsako povezavo zahteva uporabniško ime in geslo.

V nizu Connect lahko podamo uporabniško ime, če želimo, pa tudi geslo:

```
ODBC;DSN=Streznik SQL;UID=Uporabnik;PWD=geslo
```

Čeprav lahko simbolično ime podatkovnega vira že opisuje želeno zbirko podatkov, lahko spremenimo tudi to, z določilom DATABASE=, ki ga vključimo v niz Connect. Zelo koristno je lahko še eno določilo, LOGINTIMEOUT=. To določilo podaja



Slika 14-6: Gonilnik zna povprašati tudi po manjkajočem geslu in drugih parametrih.

najdaljši čas v sekundah, ki ga gonilnik porabi za vir ODBC pri vzpostavljanju povezave. Če po izteku tega časa povezava še ni vzpostavljena, se povzroči izvajalna napaka.

Povezana tabela

Z nizom Connect lahko odpremo celotno zbirko podatkov ali pa v obstoječo, domorodno zbirko priključimo zunanjo tabelo. To je mogoče z uporabo predmetov DAO ali s stavkom v jeziku Jet SQL. Priključena tabela je najbolj učinkovit način izkoriščanja oddaljenih virov ODBC prek podatkovnega stroja Jet, saj se parametri vzpostavitve povezave ob prvi priključitvi shranijo v domorodno zbirko podatkov in tako ostanejo pripravljeni za poznejša povezovanja.

Oglejmo si zgled priključitve tabele s predmeti DAO:

```
Dim db As Database
Dim td As TableDef
Dim rs As Recordset

On Error Resume Next
  ' Odpri / ustvari zbirko
  Set db = OpenDatabase("Zbirka.MDB")
  If db Is Nothing Then
    Set db = CreateDatabase("Zbirka.MDB", dbLangSlovenian)
  Else
    ' Odstrani morebitno že povezano tabelo
    db.TableDefs.Delete "Povezana tabela"
    db.TableDefs.Refresh
  End If
On Error GoTo 0

  Set td = db.CreateTableDef("Priljucena tabela")
  td.Connect = "ODBC;DSN=Streznik SQL;UID=sa;PWD=;DATABASE=vb5demo"
  td.SourceTableName = "Tabela"
  db.TableDefs.Append td
  db.TableDefs.Refresh

  Set rs = db.OpenRecordset("Prikljucena tabela", dbOpenForwardOnly)
```

```
Do Until rs.EOF
  Print rs!f1
  rs.MoveNext
Loop

rs.Close
db.Close
```

Kakor vidimo, pozna predmet razreda TableDef lastnosti Connect in SourceTableName, ki popolnoma opredelita tabelo v oddaljenem viru (ta ni nujno le vir ODBC, enako lahko priključimo poljubno tabelo iz zunanjih zbirk podatkov v podprtih oblikah zapisa). Ker tabelo nemudoma tudi uporabimo, osvežimo zbirko TableDefs. Od tu naprej je priključena tabela dostopna povsem enako kakor katerakoli druga.

Prepuščene poizvedbe

Za še učinkovitejše izkoriščanje virov ODBC je včasih koristno uporabiti *prepuščene* poizvedbe. Prepuščena poizvedba (*pass-through query*) se ne obravnava v podatkovnem stroju Jet, ampak se prepusti v obdelavo podatkovnemu strežniku neposredno. Oglejmo si značilnosti prepuščenih poizvedb. Najprej prednosti:

- Ker se obravnavajo brez vmešavanja podatkovnega stroja Jet, lahko vključujejo specifične ukaze in naprednejše zmožnosti jezika SQL (npr. shranjene procedure, omejitve vnosa, sprožilce itn.), podprte samo v podatkovnem strežniku.
- Prepuščene poizvedbe razbremenijo odjemalca, saj se v celoti obravnavajo na strežni strani. To posledično zmanjša tudi promet v omrežju. Izjemno velike prihranke si lahko obetamo pri akcijskih poizvedbah, ki posodabljajo množico zapisov.
- Prepuščene poizvedbe so edini način ustvarjanja zbirk zapisov s povezovanjem tabel iz različnih zbirk podatkov v strežniku. Vzrok za to leži v naravi povezave ODBC, ki je vedno vzpostavljena z natančno opredeljeno zbirko podatkov.

In pomanjkljivosti:

- Preklicana poizvedba je lahko delno že opravila svoje delo, kar lahko pri uporabi akcijskih poizvedb pripelje do rušenja celovitosti podatkov.
- Prepuščene izbirne poizvedbe vedno vrnejo zbirko zapisov vrste Snapshot, ki ni popravljiva.
- Zapis poizvedbe mora natančno spoštovati strežniško narečje jezika SQL, ki se lahko od strežnika do strežnika precej spreminja in lahko ne podpira vseh vgrajenih funkcij visual basica (pri uporabi običajnih poizvedb v jeziku Jet SQL se prevajanje opravi samodejno).

- Posledica zgornje pomanjkljivosti je tudi nezmožnost vzpostavitve mehanizma, ki povpraša po manjkajočih parametrih poizvedbe.

Prepuščene poizvedbe se posredujejo neposredno gonilniku izbranega vira ODBC, brez vmešavanja podatkovnega stroja Jet. Gonilnik opravi ves pogovor s podatkovnim strežnikom in vrne stanje poizvedbe (npr. morebitno sporočilo o napaki). Če gre za izbirno prepuščeno poizvedbo, se ustrezni zapisi vračajo podatkovnemu stroju Jet po potrebi in ne vsi naenkrat. Tako vedenje lahko znatno zmanjša promet v omrežju.

Najbolj priporočljiv način uporabe prepuščene izvedbe je uporaba predmeta QueryDef, kar prikazuje naslednji zgled:

```
Dim db As Database
Dim qd As QueryDef
Dim rs As Recordset

On Error Resume Next
  ' Odpri / ustvari zbirko
  Set db = OpenDatabase("Zbirka.MDB")
  If db Is Nothing Then _
    Set db = CreateDatabase("Zbirka.MDB", dbLangSlovenian)
On Error GoTo 0

  ' Ustvarimo začasno poizvedbo
  Set qd = db.CreateQueryDef("")
  ' - vir ODBC
  qd.Connect = "ODBC;DSN=Streznik SQL;UID=sa;PWD=;DATABASE=vb5demo"
  ' - šlo bo za akcijske poizvedbe
  qd.ReturnsRecords = False

  ' Poizvedba, ki ustvari arhivsko tabelo
  qd.SQL = "CREATE TABLE dbo.Arhiv_Narocil " & _
    "(id CHAR(8) NOT NULL, opis VARCHAR(50), datum DATETIME NOT NULL)"
  qd.Execute

  ' Prepis v arhiv in odstranjevanje prepisanih zapisov
  qd.SQL = "INSERT INTO Arhiv_Narocil " & _
    "SELECT * FROM Narocila WHERE datum < '1/1/1998'"
  qd.Execute
  qd.SQL = "DELETE FROM Narocila WHERE datum < '1/1/1998'"
  qd.Execute

  db.Close
```

Kakor vidimo, je izdelovanje prepuščenih poizvedb omogočeno z lastnostjo Connect, ki jo pozna tudi predmet razreda QueryDef. Ker smo v nadaljevanju uporabili ukaze SQL, ki ne vračajo zapisov, moramo ustrezno nastaviti lastnost **ReturnsRecords**. Za vse tri poizvedbe smo uporabili isti, začasni predmet razreda QueryDef. Opazimo lahko spremenjen zapis datumskih konstant. Podatkovni strežnik SQL Server razume standardizirani zapis datumskih vrednosti med enojnimi narekovaji, medtem ko Jet SQL uporablja znak #. Če ne bi šlo za prepuščeno poizvedbo, bi podatkovni stroj Jet sporočil napako že pri njeni analizi.

Oglejmo si še zgled uporabe prepuščene izbirne poizvedbe:

```
Dim rs As Recordset
  qd.ReturnsRecords = True
  qd.SQL = "SELECT * FROM Arhiv_Narocil"
  Set rs = qd.OpenRecordset(dbOpenSnapshot)
```

Zbirka zapisov v predmetu rs je sedaj na voljo kakor katerakoli druga. Pri ustvarjanju zbirke zapisov s prepuščenimi poizvedbami smo omejeni na vrsto Snapshot, ki ne omogoča popravljanja vsebine.

Posebne oblike izbirnih poizvedb lahko vrnejo več zbirk zapisov (*multiple recordset queries*). Če za sklicevanje na izid tovrstne poizvedbe uporabimo predmet razreda Recordset, imamo dostop le do prve vrnjene zbirke. Rešitev te težave je v uporabi akcijske poizvedbe SELECT INTO, ki ustvari več tabel z izidi poizvedbe. Te tabele so poimenovane po zgledu prve tabele, le da jim je priključena še zaporedna številka. Če bi poizvedba vračala tri skupine zapisov, v akcijski poizvedbi pa bi za izid določili tabelo »Izid«, bi se ustvarile tabele »Izid«, »Izid1« in »Izid2«.

S prepuščenimi poizvedbami lahko izkoriščamo zahtevnejše zmožnosti podatkovnih strežnikov, kot so npr. shranjene procedure. Njihovo klicanje je zelo podobno uporabi poizvedbe, le da namesto poizvedbe lastnosti SQL priredimo niz z imenom shranjene procedure. Če ta procedura vrača vrednost, jo lahko preberemo v prvem polju vrnjene zbirke zapisov:

```
qd.SQL = "ShranjenaProcedura"
Set rs = qd.OpenRecordset(dbOpenSnapshot)
' Izid je shranjen v prvem polju vrnjene zbirke zapisov
izid = rs(0)
```

Prepuščene poizvedbe lahko uporabimo tudi brez posebnega predmeta razreda QueryDef, kar je priročna rešitev za majhno število akcijskih poizvedb. Uporabimo kar lastnost in metodo predmeta razreda Database:

```
Set db = OpenDatabase("Zbirka.MDB")
db.Connect "ODBC;DSN=Streznik SQL;UID=sa;PWD=;DATABASE=vb5demo"
db.Execute "DELETE FROM Arhiv_Narocil", dbSQLPassThrough
```

Zbirka podatkov, ponazorjena s predmetom db, je lahko katerakoli domorodna zbirka. Prepuščena poizvedba ni v ničemer povezana z njo, je zgolj most do potrebne lastnosti in metode. Z nastavitvijo lastnosti Connect predmet db že kaže na drugo zbirko podatkov. Z uporabo tega načina ne moremo prebrati izida izbirne poizvedbe, zato tudi ni priporočljiv za proženje shranjenih procedur.

Do podatkov neposredno

Če se želimo uporabi podatkovnega stroja Jet popolnoma izogniti, lahko uporabimo poseben delovni prostor (ODBC *direct workspace*). Pri uporabi tega delovnega prostora imamo na voljo nekaj dodatnih predmetov in posebej učinkovite asinhrone poizvedbe za čimbolj učinkovito izkoriščanje podatkovnega strežnika. Pregled

večine prednosti in pomanjkljivosti neposrednega dostopa najdete na začetku tega poglavja.

Neposreden delovni prostor

Za neposreden dostop do podatkovnih virov ODBC potrebujemo poseben delovni prostor. Posredno ga lahko ustvarimo tudi z rabo gradnika Data z nastavitvijo njegove lastnosti DefaultType (1 – UseODBC). Kadar uporabimo programsko kodo, moramo poseči po privzetih lastnostih predmeta DBEngine ali pa ustvariti nov delovni prostor.

Delovni prostor za neposreden dostop do virov ODBC ustvarimo s posebno konstanto, ki jo posredujemo kot zadnji parameter pri klicu metode CreateWorkspace:

```
Dim ws As Workspace
  Set ws = DBEngine.CreateWorkspace("Prostor ODBC", _
    "Uporabnik", "geslo", dbUseODBC)
```

Ime in geslo, ki ga podamo kot obvezna parametra metode, se uporabita pri poizkusu vzpostavitve povezave, če ju ne nadomestimo z ustreznimi določili v nizu Connect.

Če se ne želimo ukvarjati z delovnimi prostori, zadošča, da spremenimo lastnost **DefaultType** predmeta DBEngine iz privzete vrednosti dbUseJet v dbUseODBC:

```
DBEngine.DefaultType = dbUseODBC
```

Prvi poizkus vzpostavitve povezave bo implicitno ustvaril privzeti delovni prostor, ki bo zaradi zgornje nastavitve samodejno predstavljal neposreden dostop do vira ODBC. Z lastnostjo **DefaultUser** in **DefaultPassword** lahko predpišemo tudi privzeto uporabniško ime in geslo, ki se uporabi pri implicitni stvaritvi *neposrednega* delovnega prostora (*direct workspace*).

Razred Connection

Pri uporabi zunanjih virov skozi podatkovni stroj Jet smo morali določiti predmet razreda Database, ki nam je rabil kot posrednik za dostop do podatkov. Pri uporabi neposrednega dostopa uporabe razreda Database ne priporočajo več. Za dostop do podatkovnega strežnika imamo na voljo poseben razred, tako imenovani **Connection**, ki ponuja nekaj zelo učinkovitih zmogljivosti:

- Asinhrono povezovanje. Namesto da programska koda čaka na vzpostavitev povezave s podatkovnim strežnikom, lahko nadaljujemo delo in uspešnost vzpostavitve preverimo kasneje.
- Asinhrono izvajanje poizvedb. Izvajanje poizvedb je mogoče naložiti strežniku in vmes nadaljevati izvajanje programske kode.

- Uporabo predmetov razreda QueryDef. Pri neposrednih delovnih prostorih teh predmetov ne moremo uporabljati kot lastnosti razreda Database.

Nov izvod predmeta ustvarimo s klicem metode **OpenConnection**. Lahko jo uporabimo v izdelanem delovnem prostoru, če pa smo za privzeto obliko delovnega prostora določili neposredno povezovanje, lahko metodo pokličemo tudi samostojno. Simbolični zapis klica metode se glasi:

```
Set povezava = [delovniProstor.]OpenConnection( _
  imeDSN, opcije, samoBranje, povezava _
)
```

Povezava je nov predmet razreda Connection, ime DSN je lahko simbolično ime določenega vira DSN ali pa predstavlja novo ime, če bomo uporabljeni vir opisali v nizu parametra povezava (niz Connect). Vrednost tega parametra hkrati poimenuje povezavo znotraj zbirke Connections. Parameter izbire določa vrsto nastavitev, ki vplivajo na potek povezovanja. Samo branje odpre povezavo v načinu, ki ne dopušča spreminjanja vsebine zapisov.

Odprta povezava omogoča neposredno izvajanje akcijskih poizvedb in uporabo predmeta QueryDef za izvajanje vseh poizvedb in tvorbo zbirk zapisov (*recordsets*). Oglejmo si zgled:

```
Dim con As Connection

  DBEngine.DefaultType = dbUseODBC
  Set con = OpenConnection("", _
    dbDriverPrompt, False, "ODBC;DSN=Streznik SQL")
  con.Execute "DELETE FROM Arhiv_Narocil"
  con.Close
```

V zgornjem zgledu smo odprli začasno povezavo, zato je nismo poimenovali. Z nastavitvijo parametra izbir smo naložili gonilniku, naj nas povpraša po manjkajočih podatkih. Po vzpostavitvi izvedemo akcijsko poizvedbo in povezavo zapremo.

Asinhrono delo

Ker je povezovanje s podatkovnimi strežniki večinoma odvisno od razmer v omrežju, lahko traja precej dolgo. Že nekaj sekund je pri računalnikih veliko časa, še posebej, če jih moramo presedeti pred navidezno nedejavnim zaslonom. Za izboljšanje odzivnosti programa in boljši nadzor nad potekom vzpostavljanja povezave lahko izberemo *asinhrono* povezovanje. Asinhrono povezovanje (*asynchronous connection*) omogoča programski kodi, da nadaljuje izvajanje tudi po izdanem zahtevku za povezovanje. Z lastnostjo **StillExecuting** preverimo, ali se vzpostavljanje povezave še ni končalo. Na voljo imamo tudi metodo **Cancel**, s katero lahko postopek povezovanja kadarkoli prekinemo. Oba elementa sta na voljo tudi pri predmetih razredov QueryDef in Recordset.

Spodnji zgled poizkusi vzpostaviti povezavo asinhrono, kar mu omogoča posebna nastavitev parametra izbir. S funkcijo Timer spremljamo čas vzpostavljanja povezave in ga prekličemo, če se ne konča v vnaprej določenem časovnem intervalu:

```
Private con As Connection

Private Sub btnPriklop_Click()
Const cPrag As Long = 10
Dim lngTick As Long
Dim intCas As Integer

  btnPriklop.Enabled = False

  DBEngine.DefaultType = dbUseODBC
  Set con = OpenConnection("Streznik SQL", _
    dbDriverNoPrompt + dbRunAsync, _
    False, "ODBC;DSN=Streznik SQL;UID=;PWD=;")

  lngTick = Int(Timer)
  intCas = 0
  Do While con.StillExecuting
    If Int(Timer) > lngTick Then
      lngTick = Timer
      intCas = intCas + 1
      Label1.Caption = intCas
      Label1.Refresh
      If intCas > cPrag Then
        Label1.Caption = "Preklic povezave!"
        Label1.Refresh
        con.Cancel
        btnPriklop.Enabled = True
        Exit Do
      End If
    End If
  Loop

End Sub
```

> **NAMIG** Če želimo zgolj časovno omejiti najdaljši čas povezovanja, lahko uporabimo običajno povezovanje in v nizu Connect podamo določilo LOGINTIMEOUT=.

Pri neposrednem dostopu do podatkovnih virov ODBC lahko asinhrono izvajamo tudi poizvedbe. Izvajanje poizvedb je mogoče z metodo Execute neposredno na vzpostavljeni povezavi, pri izbirnih poizvedbah pa je edini uporabni način tvorba izvoda razreda QueryDef. Ta ima v neposrednih delovnih prostorih nekatere posebne značilnosti:

- Vsi, tudi poimenovani predmeti razreda QueryDef, so začasni.
- Nanje se sklicujemo kot na lastnosti predmeta Connection.
- Omogočajo izvajanje asinhronih poizvedb.

Pri predmetih razreda QueryDef lahko, tako kakor v običajnem delovnem prostoru, uporabimo metodo Execute za izvajanje akcijskih poizvedb, ali metodo OpenRecordset za pregled izida izbirne poizvedbe oziroma shranjene procedure:

```
Dim qd As QueryDef
Dim rs As Recordset

  Set qd = con.CreateQueryDef("Poizvedba")
  qd.SQL = "SELECT * FROM Narocila"

  Set rs = qd.OpenRecordset(dbOpenDynamic, dbRunAsync, dbOptimistic)

  ' Delamo naprej ...

  If qd.StillExecuting Then
    qd.Cancel

  Else
    ' Uporabimo zbirko zapisov

  End If
```

Pri neposrednih delovnih prostorih nam je na voljo posebna oblika zbirke zapisov, Dynamic, katere lastnosti smo opisali, ko smo obravnavali različne vrste zbirk. Osnovna značilnost te zbirke je sprotna usklajenost z dejanskimi podatki v zbirki.

Asinhrone akcijske poizvedbe so lahko problematične. Ker se vsebina v celoti posodablja v podatkovnem strežniku, je mogoče, da prekinjena poizvedba pusti nekaj zapisov popravljenih, preostanka pa še ne. Tej težavi se izognemo tako, da asinhrono akcijsko poizvedbo uporabimo znotraj odprte transakcije:

```
BeginTrans

cnn.Execute "DELETE FROM Narocila", dbRunAsync

' Nadaljujemo delo ...

If cnn.StillExecuting Then
  cnn.Cancel
  ' Preprečimo samo delno izpeljavo operacije
  Rollback
Else
  ' Potrdimo končano operacijo
  CommitTrans
End If
```

Pri potrjevanju odprte asinhrone poizvedbe je dobro preveriti stanje lastnosti StillExecuting. Če bi se poizvedba še vedno izvajala, ko pokličemo metodo CommitTrans, bi bilo asinhronega izvajanja konec, saj bi koda počakala do konca poizvedbe in jo zatem potrdila. Učinkoviteje je večkrat preveriti stanje omenjene lastnosti in šele na koncu izdati potrditev transakcije.

Med izvajanjem asinhrone operacije je predmet razreda Connection *zaseden*. Če želimo izvajati več asinhronih poizvedb hkrati, moramo odpreti več povezav.

Paketno posodabljanje

V večini programov, ki izkoriščajo model odjemalca in podatkovnega strežnika, se vsebina zapisov posodablja postopoma, zapis za zapisom. To pa je lahko precej dolgotrajno. Poteka namreč v naslednjih korakih:

1. Uporabnik popravi vrednost zapisa in jo želi shraniti.
2. Strežnik skuša zakleniti popravljeni zapis in mu posodobiti vsebino.
3. Če ni uspešno, je obveščen program, ki sproži izvajalno napako.

Vidimo, da se uporablja optimistično zaklepanje posameznega zapisa. Pri neposrednem delovnem prostoru imamo na voljo tudi možnost *paketnega* optimističnega posodabljanja vsebine. Paketno posodabljanje (*batch optimistic updating*) nam omogoča, da v zbirki popravimo poljubno mnogo zapisov, spremembe pa shranimo naenkrat.

Naslednji zgled prikazuje uporabo paketnega obnavljanja:

```
Dim ws As Workspace
Dim con As Connection
Dim rs As Recordset

  DBEngine.DefaultType = dbUseODBC
  Set ws = DBEngine.Workspaces(0)
  ' Določi ustrezno obliko kazalca
  ws.DefaultCursorDriver = dbUseClientBatchCursor
  Set con = OpenConnection("", dbDriverNoPrompt, False, _
    "ODBC;DSN=Streznik SQL;DATABASE=pubs;UID=sa;PWD=;")
  ' Odpri popravljivo zbirko zapisov s paketnim obnavljanjem
  Set rs = con.OpenRecordset("SELECT * FROM roysched", _
    dbOpenDynaset, 0, dbOptimisticBatch)

  ' Vse spremembe vsebine se pomnijo pri odjemalcu
  Do While Not rs.EOF
    rs.Edit
    rs!royalty = rs!royalty + 5
    rs.Update
    rs.MoveNext
  Loop

  ' shrani vse spremembe še pri strežniku
  rs.Update dbUpdateBatch
```

Za uporabo paketnega obnavljanja *moramo* določiti posebno obliko *kazalca* zapisa (*cursor*), ki ga določa konstanta dbUseClientBatchCursor. Zbirko zapisov moramo odpreti s posebno konstanto, ki strežniku dopove, da bomo zapise obnavljali v paketnem, optimističnem načinu (dbOptimisticBatch). Vse spremembe, ki jih pozneje napravimo na zapisu, se hranijo pri odjemalcu. Ko želimo spremembe shraniti tudi pri strežniku, to *poizkusimo* z metodo Update, ki ji posredujemo poseben parameter, vrednost konstante dbUpdateBatch. Če bi bil v trenutku klica paketnega obnavljanja še kak zapis odprt za posodabljanje vsebine (Edit), bi se ta zapis samodejno posodobil pred paketnim obnavljanjem.

NAMIG Seveda rabi zgornji zgled le za demonstracijo paketnega obnavljanja. Mnogo elegantneje bi enako opravilo »paketno« dokončala akcijska poizvedba. Če je v popravljanje zapisov vpleten tudi uporabnik, pa se vse spremeni.

Tudi med paketnim obnavljanjem, ko se vse spremembe zapisov še hranijo pri odjemalcu, je mogoče neposredno v podatkovnem strežniku obnoviti vsebino posameznega zapisa. To obnavljanje ne vpliva na druge shranjene spremembe, ki se bodo izvedle šele s posebnim klicem metode Update. Za shranjevanje sprememb samo na trenutnem zapisu izdamo klic metode:

```
rs.Update dbUpdateCurrentRecord
```

Pri opisu zgleda paketnega obnavljanja smo omenili, da klic metode Update s parametrom v vrednosti konstante dbUpdateBatch poizkusi posodobiti popravljene zapise. Zaradi optimističnega načina zaklepanja je mogoče, da so nekateri izmed popravljenih zapisov odprti pri drugih uporabnikih. Tej nevšečnosti pravimo *kolizija* in jo je mogoče nadzorovati ter popraviti.

Po izdaji ukaza za paketno obnavljanje moramo preveriti lastnost BatchCollision predmeta razreda Recordset. Če je pri paketnem posodabljanju prišlo do kolizije (*collision*), bo v tej lastnosti shranjeno polje oznak (*bookmarks array*), ki enolično določa vse zapise, ki jih ni bilo mogoče posodobiti. V teh zapisih lahko analiziramo vsebino vsakega polja posebej s posebnimi lastnostmi predmeta razreda Field. Te so:

Lastnost	Opis
Value	Trenutna, posodobljena vrednost polja, kot je shranjena pri odjemalcu.
OriginalValue	Prejšnja vrednost polja, kakršna je bila pred posodobitvijo.
VisibleValue	Vrednost polja, kot je shranjena v podatkovnem strežniku.

Za razreševanje kolizije imamo na voljo več možnosti:

- Vsilimo našo, posodobljeno vrednost (Value) s klicem metode Update, ki mu dodamo še tretji parameter vsiljevanja sprememb (*force*).
- Obnovimo staro vrednost polja (OriginalValue) in obnovimo prejšnje stanje z vsiljenim obnavljanjem
- Zanemarimo našo spremembo in dopustimo novo, popravljeno vsebino (VisibleValue).

V programski kodi so našteta opravila videti takole:

```
  ' poizkusi shraniti vse spremembe
  rs.Update dbUpdateBatch

Dim i As Integer

  For i = 0 To rs.BatchCollisionCount - 1

    ' aktiviramo problematičen zapis
    rs.Bookmark = rs.BatchCollisions(i)

    ' pogoj določa, kaj se zgodi z vrednostjo zapisa
    Select Case pogoj

      Case cVsiliSpremembo
        rs.Update dbUpdateCurrentRecord, True

      Case cVrniPrejsnjeStanje
        With rs!Royalty
          .Value = .OriginalValue
        End With
        rs.Update dbUpdateCurrentRecord, True

      Case cZanemariSpremembe
        ' posvojimo novo vrednost, za uskladitev bo poskrbela
        ' metoda Requery spodaj.

    End Select

  Next

  ' uskladi zbirko z dejanskim stanjem
  rs.Requery
```

V zgornji kodi smo ponazorili vsako od treh možnosti z izmišljenimi konstantami. V praktičnih zgledih se za vrsto ukrepa najpogosteje odločimo na podlagi stare, obnovljene in dejanske vrednosti polja zapisa. Na koncu ne smemo pozabiti zbirke zapisa uskladiti z dejanskim stanjem, kar opravi metoda **Requery**.

> **NAMIG** Določilo, ki vsili zapis sprememb, je mogoče uporabiti tudi pri paketnem obnavljanju, vendar to ni priporočljivo, saj lahko izniči prednosti večuporabniškega dela.

Paketno obnavljanje lahko s pridom izkoristimo za *prelivanje* množice zapisov (*data pipelining*) iz ene zbirke podatkov z enako strukturo v drugo. Prelivanje ponazarja spodnji zgled:

```
Dim ws As Workspace
Dim cn1 As Connection, cn2 As Connection
Dim rs As Recordset

  DBEngine.DefaultType = dbUseODBC
  Set ws = DBEngine.Workspaces(0)

  ws.DefaultCursorDriver = dbUseClientBatchCursor
```

```
Set cn1 = ws.OpenConnection("Con1", dbDriverNoPrompt, False, _
  "ODBC;DSN=Pubs;UID=sa;PWD=;")
Set cn2 = ws.OpenConnection("Con2", dbDriverNoPrompt, False, _
  "ODBC;DSN=PubsNew;UID=sa;PWD=;")

Set rs = cn1.OpenRecordset("SELECT * FROM sales " & _
  "WHERE ord_date BETWEEN '1/1/1995' AND '12/31/1995'", _
  dbOpenDynaset, 0, dbOptimisticBatch)

Set rs.Connection = cn2

Do Until rs.EOF
  rs.RecordStatus = dbRecordNew
  rs.MoveNext
Loop

rs.Update dbUpdateBatch, True
```

Za prelivanje podatkov uporabimo navezavo izdelane zbirke zapisov na nov podatkovni vir z njeno lastnostjo Connection. Prepis podatkov potem zahteva zgolj spremembo stanja vsakega zapisa posebej. Zgornji zgled preliva podatke iz ene zbirke v drugo v istem strežniku. To opravilo je mogoče učinkoviteje opraviti s posebno poizvedbo, vendar je zgornji zgled splošno uporaben tudi za vire iz dveh popolnoma različnih strežnikov.

Značilnosti

Delo s podatkovnim strežnikom je precej odvisno od vrste strežnika, njegovih zmogljivosti in še posebej njegovih posebnih zmožnosti, ki jih najbolje izkoristimo z neposrednim delovnim prostorom. Za konec obravnave dela s podatkovnimi strežniki si oglejmo še nekaj značilnosti dela s podatki v zunanjih virih.

Povezovanje

Povezovanje s podatkovnim virom ODBC največkrat predpostavlja dostop do omrežja, ki je časovno precej potratna operacija. Podatkovni stroj Jet omogoča medpomnjenje (*caching*) informacij, ki omogočajo hitrejši priklop na vire. Ta prihranek je najočitnejši pri priključenih zunanjih tabelah (*attached tables*), saj se njihova struktura prepiše v krajevno, domorodno zbirko in so vsi poznejši priklopi znatno hitrejši.

Učinek medpomnjenja lahko dosežemo tudi s *predpovezovanjem* (*preconnecting*). To je posebna tehnika, s katero odpremo navidezno zbirko podatkov in jo nemudoma zatem zapremo. Z zapiranjem zbirke podakov ne bomo podrli vzpostavljene povezave, ki je zato hitro na voljo naslednjič. Oglejmo si zgled v programski kodi:

```
Dim strConnect
Dim dbFoo As Database
```

```
  ' začasna povezava, dokončno jo določi uporabnik
  Set dbFoo = OpenDatabase("", False, False, "ODBC;")
  strConnect = dbFoo.Connect
  dbFoo.Close

  ' ...

Dim db As Database

  ' hitro vzpostavimo povezavo, ko jo potrebujemo
  Set db = OpenDatabase("", dbDriverNoPrompt, False, strConnect)
```

Zmanjšanje potrebnih povezav

Pri delu s podatkovnimi viri je mogoče znotraj ene povezave hkrati uporabiti le eno poizvedbo. Pri uporabi storitev podatkovnega stroja Jet nas ta značilnost niti ne zanima, drugače pa je pri neposrednih delovnih prostorih, kjer moramo za povezave skrbeti sami. Težave pri povezavah so lahko še posebej očitne pri cenejših različicah zmogljivejših podatkovnih strežnikov, ki ponavadi ne omogočajo večjega števila hkratnih povezav.

Kljub temu je pri uporabi podatkovnega stroja Jet koristno poskrbeti, da zbirke zapisov vrste Dynaset – če je to mogoče – ne presegajo 100 zapisov. V tem primeru lahko podatkovni stroj nadzoruje povezavo med odjemalcem in strežnikom z eno samo povezavo. V nasprotnem primeru uporabi dve, v prvi zahteva ključe zapisov, v drugi pa vsebino zapisov, ki jih mora prikazati na zaslonu. Druge zbirke vrste Dynaset si lahko delijo drugo povezavo, prve pa ne, saj se vsi ključi morda še niso prenesli. Tako npr. tri zbirke vrste Dynaset z več kakor 100 zapisi zasedejo štiri povezave, z manj kakor 100 zapisi pa samo eno.

Drugi način zmanjšanja števila hkratnih povezav je hitro zapiranje že odprtih. Podatkovni stroj Jet zapre povezavo, brž ko prebere vse vrednosti zapisov, ki sodijo v zbirko, ali če zbirko eksplicitno zapremo. Branje vseh vrednosti dosežemo na dva načina:

- V izbirno poizvedbo jezika Jet SQL dodamo določilo TOP 100 PERCENT, ki zagotavlja branje vseh zapisov.
- V kodi visual basica v odprti zbirki zapisov pokličemo metodo MoveLast.

Čeprav oba prijema skrajšata čas, v katerem je povezava odprta, lahko znatno obremenita odjemalca, zato ju ne uporabljamo pri zelo obširnih zbirkah zapisov.

Kazalci

Pri delu z zbirkami podatkov se že od začetka srečujemo s pojmom dejavnega oziroma trenutnega zapisa. Omenili smo že, da nanj kaže podatkovni kazalec (*cursor*). Pri delu s podatkovnimi strežniki nam *kazalec* predstavlja enega ali več *enoličnih označevalcev* (*keyset*), ki pripadajo vsakemu posameznemu vrnjenemu

zapisu izbirne poizvedbe. V prejšnjih zgledih smo že uporabljali posebne nastavitve kazalca, ki omogočajo posebne načine dela, kot je npr. paketno optimistično posodabljanje zapisov. Vrsto kazalca delno določa posebna lastnost delovnega prostora, dokončno pa jo določimo pri odpiranju zbirke zapisov (*recordset*).

Vrste kazalcev

Osnovna značilnost kazalcev je, da enolično predstavljajo izbrane podatke v podatkovnem strežniku. Druge lastnosti kazalcev so:

- Predstavljajo lahko vse zapise ali izbor zapisov iz ene tabele.
- Predstavljajo lahko vse zapise ali izbor zapisov iz združene tabele (*table join*).
- Lahko ne predstavljajo nobenega zapisa.
- Lahko omogočajo spreminjanje vsebine zapisov ali pa ne.
- Lahko jih prestavljamo v obe smeri (*fully scrollable*) ali pa samo naprej (*forward only*).
- Lahko se vzdržujejo na strani odjemalca ali pa v podatkovnem strežniku.
- Lahko kažejo neposredno na dejanske podatke ali pa le na njihove ključe.

Na učinkovitost dela najbolj vpliva mesto vzdrževanja kazalca. Če je mogoče kazalce prepustiti v vzdrževanje strežniku, lahko znatno zmanjšamo promet po omrežju in s tem učinkovitost dela v večuporabniških okoljih. Slaba lastnost strežniških kazalcev je večja poraba sistemskih sredstev (hitrega in zunanjega pomnilnika) pri strežniku, zato mora biti ta pravilno dimenzioniran in prilagojen številu odjemalcev. Mesto vzdrževanja kazalca določa lastnost delovnega prostora, imenovana **DefaultCursorDriver**. Njene nastavitve so zbrane v spodnji tabeli:

Nastavitev lastnosti DefaultCursorDriver	Opis
dbUseDefaultCursor	Prepusti izbiro mesta vzdrževanja kazalca gonilniku ODBC. To je najvarnejša izbira.
dbUseLocalCursor	Vzdržuje kazalec pri odjemalcu. Ta možnost je učinkovita samo pri zelo majhnih skupinah zapisov.
dbUseServerCursor	Vzdržuje kazalec pri strežniku. Zelo učinkovita možnost, ki pa je ne podpirajo čisto vsi podatkovni strežniki.
dbUseClientBatchCursor	Posebna nastavitev pri paketnih obdelavah. Kazalci se vzdržujejo pri odjemalcu.
dbUseNoCursor	Vse zbirke zapisov se odpirajo v obliki ForwardOnly z enim samim zapisom, ki ne dopušča posodabljanja vsebine (*cursorless queries*).

Vrsta kazalca je dokončno določena z vrsto zbirke podatkov, ki jo odpremo s stvaritvijo novega izvoda razreda Recordset. Vrsto zbirke določimo z neobveznim parametrom pri klicu metode OpenRecordset. Odvisnost vrste kazalca in ustrezne zbirke zapisov je opisana v spodnji tabeli:

Vrsta zbirke zapisov	Vrsta kazalca
Dynamic	Dinamični kazalec, ki kaže neposredno na dejanske zapise in omogoča popravljanje njihove vsebine.
Dynaset	Dinamični kazalec, ki kaže na enolične ključe dejanskih zapisov in omogoča popravljanje njihove vsebine (privzeto v običajnih delovnih prostorih).
Snapshot	Statičen, dvosmeren kazalec, ki ne omogoča popravljanja vrednosti zapisov.
ForwardOnly	Statičen kazalec, ki ne omogoča popravljanja vsebine in dopušča samo pomik naprej (privzeto v neposrednih delovnih prostorih).

Zaklepanje zapisov

Pri odpiranju zbirke zapisov lahko poleg njene vrste predpišemo tudi način zaklepanja zapisov, s katerim se ugotavlja posodobljenost zapisov pri njihovem morebitnem popravljanju. Možne nastavitve so zbrane v spodnji tabeli:

Določilo zaklepanja	Opis
dbOptimistic	Posodobljenost zapisa se ugotavlja tik pred posodabljanjem s primerjanjem časovnih oznak posodobljenega in dejanskega zapisa (*concur row version*). Način optimističnega zaklepanja.
dbOptimisticValue	Posodobljenost zapisa se ugotavlja tik pred posodabljanjem na podlagi primerjanja vsebine posodobljenega in dejanskega zapisa (*concur values*). Način optimističnega zaklepanja.
dbPessimistic	Zapis se zaklene v trenutku vstopa v način posodabljanja (Edit ali AddNew) in se sprosti šele po zapisu popravka (*concur lock*). Pesimistično zaklepanje.
dbReadOnly	Popravljanje ni dovoljeno.

Pri optimističnem zaklepanju se sproži izvajalna napaka, če izid primerjave pred posodabljanjem govori v prid dejanskemu zapisu v zbirki. Pri pesimističnem zaklepanju je izvajalna napaka mogoča samo pred vstopom v način posodabljanja, potem pa je zapis zaklenjen in je njegova posodobitev zagotovo uspešna (če ne pride do zunanjega vpliva, kot je npr. izguba povezave).

Uporabo programske kode, ki pravilno obravnava izvajalne napake pri zaklepaju zapisov, smo spoznali v prejšnjem poglavju (uporaba predmeta Recordset).

Značilnosti delovnih prostorov

V delovnem prostoru domorodnih zbirk podatkov, ki jih nadzoruje podatkovni stroj Jet, je privzeta oblika zbirke zapisov vrsta Dynaset, ki dopušča največ svobode. Kazalci v tej zbirki so ključi izbranih vrednosti in omogočajo pomikanje v obe smeri ter optimistično posodabljanje dejanske vsebine.

V neposrednem delovnem prostoru za delo z viri ODBC je privzeta oblika zbirke zapisov ForwardOnly, ki ni najbolj prilagodljiva, je pa najhitrejša. Dopušča le pomik naprej in ne omogoča popravljanja zapisov. Če želimo odpreti popravljivo obliko zbirke zapisov, moramo metodi OpenRecordset posredovati parametra, ki določata vrsto popravljivega zapisa in povozita določilo dostopa samo za branje vrednosti:

```
Set rs = qd.OpenRecordset(dbOpenDynamic, 0, dbOptimistic)
```

15. POGLAVJE

Razpečevanje

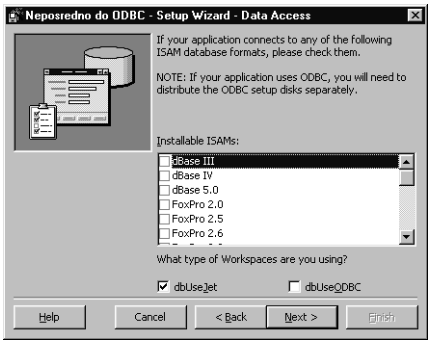
Postopek priprave namestitvene različice programa, razvitega v visual basicu, smo spoznali na koncu drugega dela knjige. Projekti, ki se ukvarjajo z zbirkami podatkov, morajo v ciljne sisteme namestiti še nekaj dodatnih sestavin.

Predmeti DAO

Čarovnik za izdelovanje namestitvenih postopkov (*Application Setup Wizard*) samodejno zazna projekte, ki vključujejo predmete podatkovnega modela DAO. To seveda ni nič posebnega, saj smo morali ustrezno knjižnico tako ali drugače vključiti v projekt. Nekaj posebnega bi bilo, ko bi čarovnik znal ugotovili oblike zapisa zbirk podatkov, ki jih uporabljamo v projektu.

Tega seveda ne zna, zato nam v posebnem koraku namestitvenega postopka postreže s seznamom, s katerega moramo izbrati uporabljene oblike zapisa. V tem koraku določimo tudi vrsto delovnih prostorov, ki jih naš program uporablja.

Slika 15-1: Poseben korak za programe z dostopom do zbirk podatkov



Na podlagi naše odločitve bo čarovnik zbral vse potrebne datoteke. Pravilno upošteva tudi vklop stikala za uporabo neposrednega delovnega prostora. Ta korak pa v namestitveni postopek *ne* vključi potrebnih gonilnikov za uporabljene vire ODBC!

Če smo v programu vzpostavili lastni varnostni mehanizem s sistemsko zbirko podatkov (npr. Varnost.MDW), je nikar ne pozabimo ročno vključiti med datoteke namestitvenega postopka.

Gonilniki ODBC

Če uporabljamo boljši različici visual basica (Professional in Enterprise), imamo na voljo gonilnike ODBC za podatkovna strežnika Microsoft SQL Server in Oracle. Z nakupom visual basica smo dobili tudi dovoljenje za razširjanje teh gonilnikov skupaj z izdelanimi programi, ki jih uporabljajo.

Za pravilno delovanje razvitega programa, ki uporablja vire ODBC, moramo v ciljnem sistemu najprej namestiti ustrezne gonilnike ODBC, šele zatem lahko namestimo tudi naš program. V nasprotnem primeru je nevarno, da se določene sestavine našega programa ne bodo pravilno vzpostavile.

Namestitveni program in potrebne datoteke za vzpostavitev priloženih gonilnikov ODBC najdemo na plošči CD-ROM visual basica v mapi \VB\ODBC\. Namestimo jih z zagonom programa Setup.EXE. Te datoteke žal ne moremo neposredno vključiti v namestitveni postopek programa, zato je treba poskrbeti za njihovo ustrezno namestitev drugače.

Nalogo nameščanja gonilnikov ODBC lahko prepustimo uporabniku ciljnega sistema, prijaznejša pa je posodobitev namestitvenega programa (projekta Setup1.VPB v mapi Setupkit\Setup1\), ki naj na začetku svojega izvajanja s funkcijo **Shell** najprej sproži namestitveni postopek gonilnikov ODBC. Žal so datoteke precej obsežne, tako da bo najboljši nosilec za razpečevanje CD-ROM ali zmogljivejši zunanji pomnilnik (npr. disketa Zip).

ČETRTI DEL

Predmeti

V TEM DELU

Uvod
Lastni razredi
Gradniki
Razpečevanje

Vsi programi v visual basicu so predmetni (*object oriented*), ni mogoče razviti programa, ki bi predmetov ne uporabljal. Ta trditev je morda presenetljiva, saj smo se predmetov doslej bolj izogibali kakor ne. Vzrok za to je v zgodovini visual basica, ki na začetku ni bil mišljen kot orodje za razvoj predmetne programske kode, ampak pripomoček za čim hitrejši razvoj uporabnih programčkov. Pri tem je bil tako učinkovit, da je lahko vse do svoje četrte večje prenove sramežljivo skrival svojo pravo, predmetno naravo.

Medtem pa je prišlo do preobrata. Visual basic je postal za Microsoft strateško pomembno razvojno orodje. Z vsako novo različico je pridobil podporo za vse najsodobnejše tehnologije, vgrajene v operacijski sistem. In ker so Okna postajala vse bolj in bolj predmetna, je postal odkrito predmeten tudi visual basic.

16. POGLAVJE

Uvod

S predmeti smo se v tej knjigi že izdatno seznanili. Vsak gradnik je predmet, spoznali smo predmetne spremenljivke, uporabljali smo vgrajene predmete, kot so npr. Screen, Err, App, zbirke podatkov smo obravnavali skozi celo kopico predmetov v modelu DAO. Nečesa pa doslej še nismo posebej omenili:

V visual basicu lahko ustvarjamo tudi lastne razrede (*classes*).

Ustvarjanje novih razredov je ključ do zahtevnejših zmožnosti visual basica, kot je predmetno programiranje, izdelovanje strežnikov za avtomatizacijo, priprava dejavnih dokumentov in, ne nazadnje, izdelovanje novih gradnikov uporabniškega vmesnika[57].

Razredi in predmeti

Besedna zveza »predmetno programiranje« marsikateremu programerju že na prvi pogled vzbuja strahospoštovanje, vendar brez pravega razloga. Edino, česar se lahko pri predmetnem programiranju bojimo, je, da nas sili v malo večji red in disciplino pri pisanju programske kode. No, to je – roko na srce – res lahko največja žrtev za programerja sploh.

Omenili smo že, da je vsak program v visual basicu v bistvu predmeten. Če smo povsem natančni, to ne drži povsem. Bliže resnici je trditev, da so programi sestavljeni iz mešane strukturirane in predmetne programske kode. Zato je v visual basicu prehod v svet predmetnega programiranja olajšan, po drugi strani pa je dopuščanje strukturirane kode tudi slabost, ki preprečuje učinkovito izkoriščanje vseh prednosti strogega predmetnega programiranja. Žal se v visual basicu vsega ne da narediti s čisto predmetno kodo, niti ne podpira vseh prvin predmetnega programiranja (npr. dedovanja), za kar se seveda lahko zahvalimo njegovim tvorcem.

Ker visual basic ni čistokrvno predmetno orodje, ni preveč smiselno razglabljati o teoretičnih podlagah in prednostih predmetno usmerjenega programiranja. Bolje je, da si ogledamo čim več zgledov njegove praktične rabe. Kdor bo skozi njih zaslutil, kaj vse mu predmeti ponujajo, naj jih čim bolj uporablja. Komur je visual basic

[57] Večina teh zmožnosti je omogočena samo v boljših različicah visual basica.

všeč, kot je bil predstavljen doslej, se mu z razvojem lastnih razredov niti ni treba ukvarjati. Res pa je, da bo precej zamudil.

Izrazoslovje

Da bi razumeli nadaljevanje, moramo utrditi pomen nekaterih izrazov, ki jih že poznamo, in spoznati nekaj novih. Njihov pomen bomo dodatno pojasnili tudi v zgledih, ki sledijo.

Osnovna značilnost *predmeta* (*object*) je, da združuje podatke in programsko kodo za njihovo rabo. Podatke imenujemo tudi *stanje*, programsko kodo pa *vedenje* predmeta. Oboje skupaj je *vmesnik* (*interface*), s katerim predmet uporabljamo. Ozrimo se naokoli, vse polno je predmetov. Denimo, televizor. Je sploh vklopljen? Kateri program spremljamo? Kakšna je jakost tona? Vse to opisuje njegovo stanje. Če pa želimo stanje spremeniti, uporabimo vmesnik, denimo daljinski upravljalec, ki televizorju naloži, naj spremeni program. Televizijskih programov ne moremo spreminjati, kakor se nam zazdi. Vedenje predmeta določa njihov obseg, v katerem jih lahko nastavljamo.

Predmet je tudi vsak gradnik uporabniškega vmesnika. Stanje so njegove *lastnosti*, vedenje pa njegove *metode*. Dodatno ga lahko spremljamo opazujoč njegove *dogodke*. Vmesnik, dogodki in začetno stanje predmeta so določeni z *razredom* (*class*), ki mu pripada. Vsak predmet sodi v neki razred, tako kakor vsaki spremenljivki pripada neki podatkovni tip. Stanje posameznih predmetov je sicer lahko različno, vendar ohranjajo skupno vedenje, določeno z razredom, ki mu pripadajo. Za predmet pravimo, da je posamezen *izvod* razreda (*class instance*).

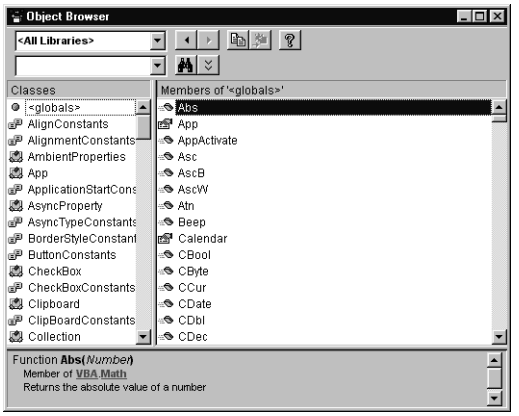
Predmete lahko uporabljamo tako, da jim dodelimo predmetno spremenljivko. Za to spremenljivko pravimo, da se na predmet *sklicuje*, oziroma da je *referenca* nanj (*object reference*). Predmetna spremenljivka je lahko predpisana določenemu razredu, npr. Database, ali pa je splošna, npr. Object. Ker spremenljivka ne vsebuje predmeta, temveč samo kaže nanj, lahko več spremenljivk predstavlja isti predmet. Zelo pomembno je vedeti, da predmet in vsi z njim povezani viri obstajajo tako dolgo, kakor dolgo obstaja kaka spremenljivka, ki kaže nanj (njihova *referenca*)!

V visual basicu so razredi združeni v *razredne knjižnice* (*type libararies*). Nekaj takih knjižnic sestavlja osnovno ogrodje programskega jezika, druge pa lahko naknadno vključujemo v projekt, tako kakor smo storili pri uporabi predmetnega modela DAO. Vsebino razrednih knjižnic lahko pregledujemo z zelo priročnim orodjem, *predmetnim brskalnikom*.

Brskalnik

Predmetni brskalnik (*object browser*) je eno izmed orodij, ki so na voljo v razvojnem okolju visual basica. Uporaben je pri vsakem projektu, še posebej prav pa nam pride pri analizi razrednih knjižnic. Prikličemo ga z izbiro ukaza **Object Browser** iz glavnega menuja **View**.

Slika 16-1: Predmetni brskalnik

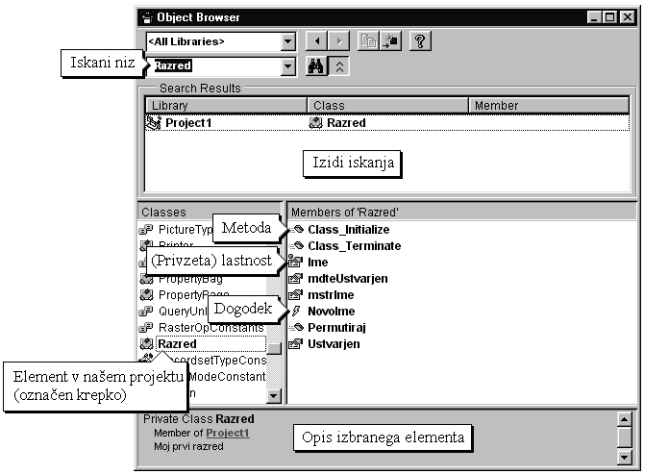


Brskalnik nam omogoča pregledovanje vseh programsko dostopnih delov knjižnice in posameznih razredov, ki so vanjo vključeni. Tu najdemo lastnosti, metode, dogodke, vgrajene konstante in še kaj.

Z zgornjega zaprtega seznama okna brskalnika lahko izberemo razredno knjižnico oziroma projekt in s tem omejimo nabor elementov, ki bi jih radi pregledovali. Na začetku je izbran prikaz iz vseh knjižnic (»<all libraries>«). Na seznamu lahko najdemo vsako posamezno predmetno knjižnico, ki je vključena v naš projekt, in tudi projekt sam, s svojimi obrazci in moduli. Na levem odprtem seznamu so našteti elementi v izbrani knjižnici, desni seznam pa našteva njegove *člane*. Člani (*members*) so vsebovani v izbranem elementu na levi. Pri seznamu elementov levo je prva vrstica rezervirana za javni del izbrane knjižnice (»<globals>«).

Namesto zamudnega brskanja po seznamu lahko uporabimo tudi iskanje. Iskani niz vnesemo v kombiniran seznam pod zaprtim seznamom knjižnic. Izid iskanja in razlago večine elementov okna prikazuje slika 16-2.

Vsi elementi, ki so razviti s kodo v našem projektu, so označeni polkrepko. Priročni menu elementov ponuja ukaz **View Definition**, ki nas prestavi v ustrezno okno s kodo, kjer je element določen. V seznamu članov na desni strani je vsak element označen s simbolom, ki opredeljuje njegovo vrsto. Opazimo lahko, da je privzeta



Slika 16-2: Predstavitev elementov v brskalniku

lastnost oziroma metoda označena z modrim krožcem poleg simbola elementa. Večina vgrajenih elementov ima krajše opise, ki se prikažejo na dnu okna brskalnika, z ukazom **Help** iz priročnega menuja pa lahko prikličemo tudi izdatnejšo sprotno pomoč.

NAMIG Po privzeti nastavitvi je okno brskalnika razredov samostojno in ne omogoča sidranja (docking). Če želimo, ga lahko postavimo ob katero od stranic glavnega okna z vklopom izbire **Dockable** v njegovem priročnem menuju. To nastavitev je mogoče določiti tudi pri preoblikovanju videza vmesnika prek ukaza **Tools ➪ Options**, jeziček **Docking**.

---

17. POGLAVJE

Lastni razredi

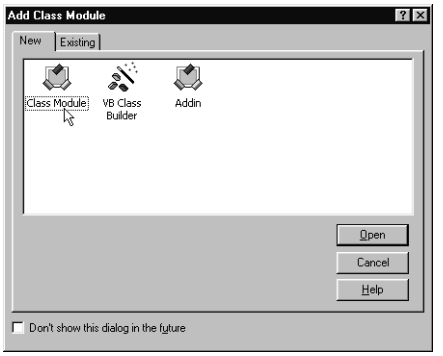
Razvoj lastnih razredov je opravilo, po katerem se ta del knjige razlikuje od dosedanjih. Z njimi lahko izkoristimo nekatere prednosti predmetnega programiranja in posežemo po zahtevnejših zmožnostih visual basica, kot je izdelovanje gradnikov ActiveX. No, če smo čisto natančni, smo svoj prvi razred razvili že s prvim zgledom v visual basicu, kajti tudi vsak obrazec je konec koncev – razred.

Razredni modul

V nadaljevanju bomo korak za korakom razvili lasten razred in ga uporabili za demonstracijo vedenja predmetov. Za razvoj programske kode najprej odpremo nov, prazen projekt vrste »Standard EXE«.

Nov razred določimo v posebnem modulu s kodo, imenovanem *razredni modul*. Razredni modul (*class module*) dodamo v projekt z izbiro ukaza **Add Class Module** iz glavnega menuja **Project**. Prikaže se seznam pripravljenih predlog. V boljših različicah visual basica imamo na voljo posebno orodje, ki nam olajša pripravo skeleta novega razreda (VB Class Builder). Spoznali ga bomo pozneje. Za nadaljevanje dela izberimo nov, prazen razredni modul (**Class Module**).

Slika 17-1: Dodajanje novega razrednega modula



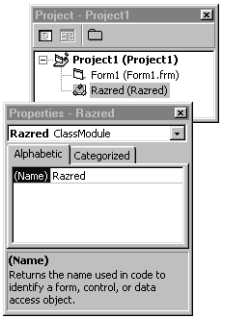
Nov modul je ponazorjen v projektnem oknu s posebnim simbolom. Podobno kakor samostojni modul s kodo tudi razredni modul ne pozna uporabniškega vmesnika,

vsebuje lahko le programsko kodo. Nov, čeprav prazen razredni modul je že viden v predmetnem brskalniku. Skupaj z obrazcem predstavlja razrede, razvite v našem projektu.

Ime

Vsakemu elementu projekta (modulu) v visual basicu pripada ime, s katerim se je mogoče nanj sklicevati v programski kodi. Ime je zelo pomembno za vsak razred, saj pravzaprav predstavlja ime novega, posebnega podatkovnega tipa, ki ga bomo lahko predpisali predmetnim spremenljivkam. Ime razrednega modula določimo v oknu za nastavljanje lastnosti, kjer je na voljo ena sama lastnost, nam že dobro znani »(Name)«. Poimenujmo naš novi razred z nič kaj domiselnim imenom »Razred«.

Slika 17-2: Poimenovanje razreda



Vmesnik

Ko določamo dela in naloge razreda, moramo najprej določiti njegov *vmesnik*. Vmesnik (*interface*) je nabor lastnosti in metod razreda, s katerimi se pogovarja z zunanjim svetom. To je izpostavljeni del razreda. Pri razvijanju programov, ki načrtno izkoriščajo prednosti predmetnega programiranja, je prav definicija ustreznega vmesnika razredov najzahtevnejše opravilo. Zelo podobno je bilo tudi pri delu z relacijskimi zbirkami podatkov, kjer je od načrtovanja tabel odvisna učinkovitost njihove rabe. Metodam načrtovanja vmesnikov pravimo *predmetna analiza* (*object analysis*).

Sama izvedba zamišljenega vmesnika pa v visual basicu ni posebej zahtevna. Določanje metod je zelo preprosto opravilo:

- Metoda je javni podprogram modula.

Pri lastnostih imamo več možnosti:

- Lastnost, dostopna brez omejitev, je javna spremenljivka modula.
- Druge lastnosti moramo določiti z eno ali več posebnimi podprogrami.

Kakor vemo že iz uporabe gradnikov in drugih predmetov, so nekatere lastnosti na voljo le za branje, druge vsiljujejo posebne omejitve, redke pa so na voljo celo samo za nastavljanje.

Življenjska doba

Naš novi razredni modul bomo dopolnili s programsko kodo, ki mu bo določila vmesnik. Naš razred bo zgolj ponazarjal življenjski cikel predmetov, tako da bi ga lahko zapisali celo brez posebnih lastnosti in metod. Zaradi boljšega razumevanja pa ga opremimo z lastnostjo, ki bo hranila ime predmeta in uslužnostno metodo.

Podobno kakor smo pri delu s podrejenimi obrazci MDI v programski kodi izdelali le prototip okna, ki smo ga lahko razmnožili v več izvodih, tudi pri razrednem modulu izdelamo prototip predmeta, kar je še eno ime za – razred. Lastnosti, določene z razredom, bodo v svojem izvodu pripadale vsakemu njegovemu predmetu.

Najprej pripravimo razred, ki ne bo poznal posebnih omejitev pri poimenovanju. V razredni modul dodamo naslednjo kodo:

```
' Razredni modul Razred

Public Ime As String

Public Sub PredstaviSe()
  MsgBox Ime
End Sub
```

Nič takega, česar bi že ne poznali. Naš razred sedaj pozna lastnost Ime, ki je niz znakov, dostopen brez omejitev, in metodo PredstaviSe, ki prikaže okno z imenom predmeta.

O življenjski dobi predmetov smo govorili že večkrat, vendar imamo šele zdaj priložnost, da si jo ogledamo na lastnem zgledu. Za preizkus razreda se vrnimo h glavnemu obrazcu. Vmesniku dodajmo ukazni gumb, ki naj ga spremlja odzivna procedura, v kateri bomo ustvarili dva izvoda razreda oziroma predmeta:

```
' Obrazec Form1

Private Sub Command1_Click()

Dim p1 As Razred
  ' Ustvarimo nov izvod razreda
  Set p1 = New Razred
  p1.Ime = "Lojze"

Dim p2 As New Razred
  ' Nov izvod se izdela samodejno
  p2.Ime = "Micka"

  p1.PredstaviSe
  p2.PredstaviSe

End Sub
```

Ko kliknemo gumb, izdelamo dva izvoda razreda Razred, vsakemu priredimo njegovo ime in ju predstavimo. Povsem enako kakor pri obrazcih lahko razred ustvarimo eksplicitno, z uporabo operatorja New, ali pa že ob napovedi predpišemo implicitno ustvarjanje novega izvoda ob prvem sklicevanju nanj. Kakor smo omenili že v prvem delu knjige, implicitnega ustvarjanja novega izvoda z operatorjem New ne moremo uporabiti za nič drugega kakor za *lastne* razrede ali zbirke, pa še tedaj ni priporočljivo. Bolje je ustvarjati nove izvode eksplicitno, v samostojnem prireditvenem stavku Set.

V razredu Razred zaenkrat nismo posebej spremljali življenjske poti predmetov, vendar bomo njegovo kodo dopolnili. Življenje vsakega predmeta se začne z dogodkom Initialize, ki ga poznamo že iz obravnave obrazca. Tudi v razrednem modulu lahko zapišemo njegovo odzivno proceduro. Prestavimo se v okno s kodo razrednega modula ter s seznama elementov izberemo **Class**, z desnega seznama dogodkov pa **Initialize**.

Slika 17-3: Izbiranje dogodkov razreda

Odzivna procedura bo zelo preprosta:

```
Private Sub Class_Initialize()
  MsgBox "Ustvarjen sem!"
End Sub
```

Konec življenjske poti predmeta označuje dogodek Terminate. Njegova odzivna procedura tudi ni posebej zapletena:

```
Private Sub Class_Terminate()
  MsgBox Ime & " se poslavlja..."
End Sub
```

Če tako dopolnjen projekt poženemo, bomo po kliku gumba najprej prejeli obvestilo o dveh stvaritvah, zatem se bosta oba predmeta predstavila, na koncu pa tudi poslovila. Večkrat smo že omenili, da predmeti končajo svoj obstoj tedaj, ko nanje ne kaže več nobena predmetna spremenljivka. V preizkusni proceduri se je zgodilo natančno to. Ker sta bili predmetni spremenljivki p1 in p2 napovedani znotraj podprograma, tudi obstajata samo znotraj podprograma. Po njegovem izteku ne obstajata več, kar pomeni tudi smrt predmetov, na katere se sklicujeta.

Sedaj pa preizkus posodobimo. Obrazcu dodajmo novo, modulsko spremenljivko tipa Razred in odziv na dogodek Load:

```
Private predmet As Razred

Private Sub Form_Load()
  Set predmet = New Razred
  predmet.Ime = InputBox("Imenuj predmet!")
End Sub
```

Dodajmo mu še en, nov ukazni gumb, katerega odziv na dogodek Click naj bo povsem enak zgornji proceduri:

```
Private Sub Command2_Click()
  Set predmet = New Razred
  predmet.Ime = InputBox("Imenuj predmet")
End Sub
```

Preizkusimo naš zgled. Takoj po zagonu, še pred prikazom obrazca, nas pozdravi sporočilo o stvaritvi novega predmeta, takoj zatem pa nas funkcija InputBox vpraša po njegovem imenu. Povsem pričakovano. Po prikazu obrazca lahko kliknemo nov gumb, ki želi vnovič vzpostaviti predmet. Kaj se zgodi? Najprej dobimo sporočilo o stvaritvi novega predmeta, zatem prvi predmet umre, na koncu pa nas znova pričaka pogovorno okno. Zakaj je umrl prvi predmet?

Modulska spremenljivka predmet kaže na izvod razreda Razred. Ob stvaritvi obrazca izdelamo tudi nov izvod razreda in ga priredimo spremenljivki. V odzivni proceduri klika gumba izdelamo nov izvod razreda. Ker tudi na nov izvod kaže ista modulska spremenljivka, prvi predmet ostane brez lastne spremenljivke in zato umre. Zakaj pa niso umirali tudi podrejeni obrazci pri urejevalniku iz drugega dela knjige? Odgovor bomo spoznali skupaj z življenjsko zgodbo obrazcev v nadaljevanju. Če nas tovrstno implicitno poslavljanje predmeta moti, lahko predmetno spremenljivko ročno postavimo na neveljavno vrednost in s tem predmet sprostimo:

```
Set predmet = Nothing
Set predmet = New Razred
predmet.Ime = InputBox("Imenuj predmet")
```

Oglejmo si prikaz še ene značilnosti, ki jo moramo upoštevati pri rabi predmetov. Če izvajanje programa končamo vsakdanje, z zapiranjem okna, nas pričaka tudi sporočilo o slovesu nazadnje ustvarjenega predmeta. Kaj pa, če bi uporabili stavek End ali nasilno prekinitev z ukazom v razvojnem okolju? Preizkusimo še ta zgled s tretjim ukaznim gumbom, ki bo nasilno prekinil izvajanje:

```
Private Sub Command3_Click()
' Nasilen konec izvajanja
  End
End Sub
```

Če izvajanje tako prekinemo, se dogodek Terminate ne bo pripetil. Izvajanje programa se prekine nemudoma, na mestu, kjer smo izdali ukaz End! Tako vedenje je zelo neprimerno pri uporabi predmetov, ki v svojem življenju zasežejo določena sredstva (npr. pomnilnik, datoteke, povezavo s podatkovnim strežnikom itn.) in bi jih morali na koncu sprostiti. Vsakemu predmetu moramo dopustiti možnost, da pospravi za seboj z odzivom na dogodek Terminate, zato moramo delo končati s stavkom Unload.

NAMIG V našem zgledu smo spremljali življenjsko pot predmetov, zato smo znotraj odzivnih procedur dogodkov Initialize in Terminate uporabili vgrajeno funkcijo MsgBox. Na splošno se moramo znotraj teh dogodkov izogibati funkcij, ki zahtevajo posredovanje uporabnika. Če želimo izpisovati določena sporočila, je mnogo bolje uporabiti vgrajeni predmet Debug in njegovo metodo Print.

Programsko kodo, razvito v tem delu knjige, najdete na priloženi plošči CD-ROM v mapi \Primeri\4-Razredi\.

Življenje gradnikov

Obrazci so posebna vrsta razrednih modulov, ker jim pripada tudi uporabniški vmesnik, hkrati pa morajo ohranjati združljivost s staro kodo. Zato se vedejo precej drugače kakor navadni razredni moduli. Razlika se zelo očitno kaže ravno pri življenjskem krogu obrazca. Obrazec ne bo izginil iz pomnilnika preprosto zato, ker se nanj ne sklicuje nobena spremenljivka več. Visual basic samodejno vzdržuje seznam vseh obrazcev v projektu. Popolnoma ga lahko odstranimo iz pomnilnika le s stavkom Unload. Zelo podobno se vede še ena vrsta predmetov, gradniki uporabniškega vmesnika.

Življenjski cikel obrazca obsega posebna stanja, ki jih pri drugih predmetih ni. Skupaj je lahko v kar štirih:

1. Ustvarjen, a ne naložen.
2. Naložen, a ne prikazan.
3. Prikazan.
4. Odstranjen.

Tako kakor pri drugih predmetih se življenje obrazca začne z dogodkom Initialize. V tem trenutku obrazec obstaja kot predmet, vendar nima vzpostavljenega okna, niti nobenega gradnika v njem. Pravimo, da je obrazec ustvarjen, vendar še ni *naložen* (*loaded*). To je edino stanje, v katerem je zagotovo vsak ustvarjeni obrazec.

Če se izvajanje programa začne z obrazcem, se njegov prvi izvod ustvari samodejno in po prvem stanju nemudoma preide v drugo in tretje. Drugače pa je z novimi izvodi. Oglejmo si zgled:

```
Dim frm As Form1
Set frm = New Form1
```

V tem stanju se obrazec, ki ga ponazarja spremenljivka frm, prav nič ne razlikuje od predmeta, določenega z razrednim modulom. Denimo, da bi imel obrazec določeno javno proceduro Pozdrav:

```
Public Sub Pozdrav()
  MsgBox "Oglašam se iz obrazca Form1!"
End Sub
```

in posodobimo zgoraj zapisani zgled:

```
Dim frm As New Form1
frm.Pozdrav
Set frm = Nothing
```

V tem primeru bi obrazec prispel le do prvega stanja, pozdravil navzoče in se takoj zatem poslovil. Uporabljamo lahko vse javne podprograme in spremenljivke, ki smo jih dodali s programsko kodo. Brž ko posežemo po kakem gradniku ali pokličemo vnaprej pripravljeno metodo obrazca, pa samodejno preide v naslednje stanje:

```
Dim frm As New Form1
frm.Pozdrav
frm.Command1.Value = True
```

Ker smo uporabili lastnost gradnika na obrazcu, lahko bi rekli kar lastnost *obrazca*, smo izsilili prehod v drugo stanje, ko je obrazec naložen, vendar še ni prikazan. Izvede se morebitna odzivna procedura dogodka Load. V zgornjem primeru smo poskrbeli tudi za to, da se pokliče odzivna procedura klika ukaznega gumba.

Če ne bi poklicali metode Show ali postavili lastnosti Visible na resnično vrednost, bi obrazec lahko ostal neviden vse do konca. To lahko opazimo, ko poizkušamo pravilno končati izvajanje programa. Glavni obrazec se sicer zapre, toda v pomnilniku ostane naložena popolna podoba njegovega drugega izvoda, ki pa nikoli ni postala vidna. Izvajanje programa moramo končati nasilno.

> **NAMIG** V starejših različicah visual basica je bila uporaba nikoli vidnih obrazcev pogosta. Program smo pač lahko končali nasilno, s stavkom End. Sedaj moramo biti pri uporabi nevidnih obrazcev previdnejši in poskrbeti, da se vsi odstranijo pravilno, s klicem stavka Unload.
>
> Še bolje je pri storitvah, ki ne potrebujejo uporabniškega vmesnika, razviti strežnike ActiveX. To opravilo bomo spoznali v nadaljevanju.

V običajno in najbolj uporabno stanje torej pridemo kličoč metodo Show. Metoda poskrbi tudi za morebitno stvaritev obrazca. Če je obrazec že ustvarjen, lahko uporabimo tudi lastnost Visible. Obrazec lahko skrijemo z metodo Hide, pri čemer preide v drugo stanje, ali pa ga odstranimo s stavkom Unload.

Kaj se zgodi po klicu stavka Unload, je odvisno od predmetnih spremenljivk. Če na obrazec ne kaže nobena več, se popolnoma odstrani. To je končno, četrto stanje. Ko na obrazec še kaže kaka predmetna spremenljivka, pa se vrne v prvo stanje: ustvarjen je, vendar ne naložen. Sprosti se njegova vizualna podoba, skupaj z vsebovanimi gradniki in morebitnimi statičnimi spremenljivkami znotraj *odzivnih* procedur. Lastne procedure in javne spremenljivke pa so še vedno na voljo.

Predmetna spremenljivka, na katero pri delu z obrazci pogosto pozabimo, je samodejno ustvarjena spremenljivka z enakim imenom kakor razred obrazca. Kakor smo že spoznali pri obravnavanju predmetov v prvem delu, visual basic za vsak obrazec samodejno doda skrito napoved. Če se razred obrazca (njegova lastnost Name) imenuje npr. Form1, se ta napoved glasi:

```
Dim Form1 As New Form1
```

Če smo se v programski kodi kadarkoli sklicevali na ime obrazca, smo posredno povzročili implicitno stvaritev in spremenljivka Form1 je še ena *referenca* obrazca. Če sedaj želimo res popolnoma odstraniti obrazec iz pomnilnika, ne smemo pozabiti na sprostitev te predmetne spremenljivke, kar dosežemo s prirejanjem vrednosti Nothing:

```
Set Form1 = Nothing
```

Poleg vseh opisanih stanj, je obrazec lahko še v enem, posebej čudnem stanju. Če smo v programski kodi shranili sklic na kak gradnik in želimo obrazec, ki ga vsebuje, popolnoma odstraniti iz pomnilnika, bomo znova prispeli v prvo stanje. To ponazarja naslednji zgled:

```
Dim Form1 As New Form1
Dim obj As Object
frm.Show
Set obj = frm.Command1
Unload frm
Set frm = Nothing
```

Brž ko bi se sklicevali na katero izmed lastnosti ali metod predmeta obj, bi obrazec samodejno prešel v drugo stanje.

> **NAMIG** Nekatere starejše različice visual basica v takih primerih niso obnovile obrazca.

Razredni moduli proti običajnim

Razredni moduli se bistveno razlikujejo od samostojnih modulov s kodo (*code module*). Vsi javni deli kodnega modula so globalno dosegljivi ves čas izvajanja programa. Razredni modul določa predlogo bodočih predmetov. Njegovi javni deli niso dostopni, dokler ne ustvarimo kake predmetne spremenljivke, ki se sklicuje nanj. In bistvena razlika: predmetov, izdelanih po vzoru razreda, je lahko več in vsi so med seboj neodvisni. Javne spremenljivke modula s kodo vedno obstajajo le v enem izvodu.

Statična lastnost razreda

Ker so predmeti med seboj neodvisni, ne vedo ničesar o zasebnem stanju drugih izvodov. Tudi oni morajo med seboj komunicirati prek objavljenega vmesnika:

lastnosti in metod. A včasih bi bilo treba kak podatek ohraniti v vseh izvodih. Sprememba njegove vrednosti v kateremkoli predmetu bi se morala odraziti v vseh. Taki vsebini pravimo tudi *statična* lastnost razreda (*static class data*), ker je enaka za vse izvode.

V visual basicu nimamo posebnega določila, s katerim bi posamezen element razreda označili kot statičen. Predlog Static je (zaenkrat) na voljo le za spremenljivke znotraj podprogramov. Statične lastnosti lahko le posnemamo. Pri tem nam pride prav samostojen modul s kodo in njegova javna spremenljivka, ki jo beremo in posodabljamo s kodo v razrednem modulu. Seveda se moramo uporabi te javne spremenljivke izogibati povsod drugje v programski kodi.

NAMIG Tak pristop tudi ni najbolj uporaben pri razvoju gradnikov ActiveX, ki ga bomo še spoznali. Pri gradnikih ActiveX lahko uporabimo *prijateljske* podprograme. Prijateljski podprogram (*friend subroutine*) je viden v razrednih modulih znotraj istega projekta, ni pa viden navzven.

Skrivanje podrobnosti

Prvi razredni modul, ki smo ga razvili, vsebuje res najpreprostejšo obliko razreda, ki ne izkorišča skoraj nobene prednosti predmetnega programiranja. Spomnimo se, da je predmet skupek podatkov in programske kode. Smisel razvijanja lastnih razredov je ravno v tem, da podatke povežemo s programsko kodo tako, da ta skrbi za njihovo pravilnost.

Oglejmo si zgled predmeta, ki bi moral opisovati stanje na bančnem računu. Če bi njegov razred razvili po vzoru razreda Razred, bi morali stanje izpostaviti kot javno spremenljivko. Uporabniki bančnega računa bi lahko z vsoto delali, kar želijo. Namenoma ali po pomoti bi lahko stanje brez težav spustili v globoko negativno vrednost.

Največ prednosti predmetnega programiranja izvira iz *skrivanja podrobnosti*. Skrivanje podrobnosti (*encapsulation*) omogoča razvoj »črnih škatel«: predmetov z znanim vmesnikom, katerega notranje delovanje nam je popolnoma skrito (in nas tudi ne zanima). Mar vemo, kaj se v resnici zgodi, ko pokličemo preprost stavek:

```
Form1.Show
```

Ne, saj nas to niti ne zanima. Pomembno je, da dosežemo končni učinek: obrazec se prikaže na zaslonu. Največja prednost, ki jo dosežemo s skrivanjem podrobnosti, je večja trdoživost (*robustness*) kode. Dobro načrtovan predmet bi moral preživeti tudi najbolj nemogoče okoliščine rabe.

Lastnosti

Trdoživost lahko le delno zagotovimo s podatkovnim tipom, ki ga predpišemo javni spremenljivki. Mnogo bolje je spremenljivko obdati s ščitom programske kode, ki preverja in ustrezno ukrepa, če pride do zlorabe. To lahko dosežemo z rabo *podprogramskih lastnosti*. Podprogramska lastnost (*property procedure*) je določena z zasebno spremenljivko in eno ali več javnimi procedurami, ki nastavljajo oziroma vračajo njeno stanje.

Za zgled bomo razvili nov razred, z imenom »Racun«. Njegova naloga bo čim popolneje nadzirati bančni račun. V projekt dodamo nov razredni modul in ga poimenujemo Racun. Za začetek moramo določiti lastnost, ki bo opredeljevala vrsto računa. Dopuščali bomo vpogledno vlogo, žiro račun z določenim limitom in kreditni račun brez omejitve dviga.

Za začetek določimo razred, ki bo vseboval vse potrebne elemente, ki opisujejo vrsto računa. Programski izpis kode v razrednem modulu Racun se glasi:

```
' Javni naštevni tip za lažje kodiranje in uporabo razreda
Public Enum VrsteRacuna
  vrVpogled = 1
  vrTekoci
  vrKredit
End Enum

' Zasebna spremenljivka, ki opisuje stanje predmeta
Private mvrVrsta As VrsteRacuna

' Javna funkcija, ki omogoči *branje* lastnosti Vrsta
Public Property Get Vrsta() As VrsteRacuna
  ' vrnemo stanje zasebne spremenljivke
  Vrsta = mvrVrsta
End Property

' Javna procedura, ki omogoča *nastavitev* lastnosti Vrsta
Public Property Let Vrsta(ByVal vrNovaVrsta As VrsteRacuna)
  ' samo, če se je vrsta spremenila
  If vrNovaVrsta <> mvrVrsta Then
    Select Case vrNovaVrsta
      Case vrVpogled, vrTekoci, vrKredit
        ' ustrezna vrsta
      Case Else
        ' neustrezna vrsta, sproži napako
        Err.Raise _
          Description:="Neveljavna vrsta računa", _
          Number:=vbObjectError + 1001
    End Select
    ' posebno pravilo
    If vrNovaVrsta < mvrVrsta Then
      Err.Raise _
        Description:="Ne morem zmanjšati vrste računa", _
        Number:=vbObjectError + 1002
```

```
    Else
      ' vse je v redu, shranimo novo vrsto
      mvrVrsta = vrNovaVrsta
    End If
  End If
End Property

' Vsi razredi z zasebnimi spremenljivkami
' stanja bi morali imeti vzpostavitev
Private Sub Class_Initialize()
  ' določimo začetno vrsto računa
  mvrVrsta = vrVpogled
End Sub
```

Preglejmo zgled od začetka. Najprej napovemo javni naštevni tip, ki opredeli vse različne vrste računa. Skrivanje podrobnosti se začne z zasebno spremenljivko mvrVrsta. Vrste računa ne bomo izpostavili kot javno spremenljivko, temveč skozi podprogramsko lastnost Vrsta. Ta bo kombinacija omenjene zasebne spremenljivke in kode, ki vsiljuje omejitve pri njeni rabi. Za branje lastnosti zapišemo poseben funkcijski podprogram Property Get, za nastavljanje pa proceduro Property Let.

Branje ni nič posebnega, preprosto vrnemo vsebino zasebne spremenljivke. Drugače je pri nastavljanju. Tu najprej preverimo, ali se nova nastavitev sploh razlikuje od shranjenega stanja. To preverjanje poveča učinkovitost kode, vendar zahteva vzpostavitev začetnega stanja predmeta, ki ga bomo morali zapisati v odzivu na dogodek Initialize. V nadaljevanju nastavljanja preverimo pravilnost nove vrednosti in dodamo še omejitev, ki preprečuje nazadovanja vrste računa. Če je vse v redu, popravimo vsebino zasebne spremenljivke, če pride do zlorabe, pa sprožimo lastno izvajalno napako. Kakor smo že omenili, je za pravilno delovanje programa pomembna vzpostavitev vsakega predmeta, izdelanega po vzoru razreda Racun. Zapisati moramo odzivno proceduro dogodka Initialize, v katerem določimo začetno stanje predmeta. Če želimo pisati trdoživo programsko kodo, moramo to proceduro zapisati pri vsakem razredu, ki uporablja zasebne spremenljivke. Z njim nedvoumno določimo začetno stanje novo izdelanega izvoda.

> **NAMIG** Pri obravnavanju pravil poimenovanja gradnikov in spremenljivk v drugem delu knjige smo omenili posebni predponi g in m, ki določata globalni oziroma modulski doseg. V dosedanjih zgledih ju nismo uporabljali, pri delu z razrednimi in kodnimi moduli pa nam prideta prav, zato ju bomo uporabili v zgledih v tem delu. Ime mvrVrsta označuje zasebno spremenljivko uporabniško določenega tipa s predpono vr (vrsta računa).

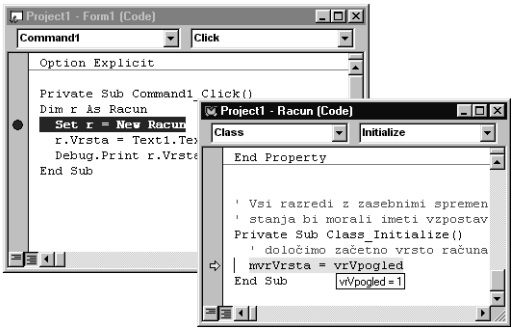
Zgoraj zapisani zgled jasno ponazarja, zakaj razredom včasih rečemo tudi »pametni« podatkovni tipi. V njih ne samo shranjujemo zbirke podatkov, temveč tem podatkom omogočamo, da se odzovejo na morebitno zlorabo. Za preizkus razreda se preselimo k obrazcu, ki mu dodamo vnosno polje (Text1) in gumb (Command1), katerega odzivna procedura dogodka Click se glasi:

```
Private Sub Command1_Click()
Dim r As Racun
  Set r = New Racun
  r.Vrsta = Text1.Text
  Debug.Print r.Vrsta
End Sub
```

Razhroščevanje

Če želimo izvedeti več o prirejanju in branju vrednosti lastnosti, je zelo koristno uporabiti vgrajeni razhroščevalnik (*debugger*). S prekinitvenimi točkami in koračnim izvajanjem lahko sledimo implicitnim klicem programske kode razrednega modula.

Naš zgled bomo razčlenili s koračnim izvajanjem z vstopom v podprograme (*step into*). Zgled poženemo z izbiro ukaza **Step Into** iz glavnega menuja **Debug**. Po zagonu programa v vnosno polje vnesemo številko 1 in kliknemo ukazni gumb. Zaradi koračnega izvajanja se nemudoma vrnemo v dejavno vrstico programske kode. Izvajanje nadaljujemo z vnovično izbiro ukaza Step Into (hitro ga dosežemo z bližnjico F8).



Slika 17-4: Prekinitvena točka, koračno izvajanje in pregledovanje vrednosti

Opazimo, da vzpostavitev novega razreda pomeni implicitni klic podprograma Initialize. Prirejanje nove vrednosti lastnosti nemudoma preskoči v podprogram Property Let, branje njene vsebine pa pomeni skok v podprogram Property Get.

Poizkusimo sedaj še z neustrezno vrednostjo, denimo 0. Koračno izvajanje nas počasi pripelje do stavka Raise, ki sproži izvajalno napako zaradi poizkusa prirejanja neustrezne vrednosti lastnosti.

Slika 17-5: Izvajalna napaka zaradi neustrezne vrednosti lastnosti

Posodobimo sedaj zgled s prestrezovalnikom izvajalne napake, ki se ustrezno odzove ob nepravilno določeni vrednosti lastnosti vrsta:

```
Private Sub Command1_Click()
Dim r As Racun
  Set r = New Racun

On Error GoTo Napaka
  r.Vrsta = Text1.Text
  Debug.Print r.Vrsta
  Exit Sub

Napaka:
  MsgBox "Napačna vrednost!"
  Resume Next

End Sub
```

Odstranimo prej določeno prekinitveno točko (z enakim postopkom, kakor smo jo postavili) in vnovič sprožimo izvajanje programa. Spet bomo preizkusili vedenje ob vnosu nepravilne vrednosti, npr. 4. Vedenje vgrajenega razhroščevalnika je sedaj odvisno od posebne nastavitve v razvojnem okolju. Lahko se pripeti dvoje:

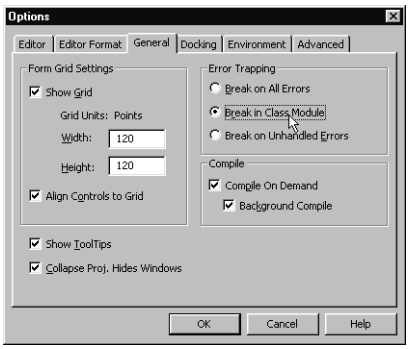
- Oglasi se prestrezovalnik napake, ki opozori na nepravilno vrednost in nadaljuje izvajanje programa.
- Kljub našemu prestrezovalniku se prikaže sporočilo o izvajalni napaki.

Na vedenje vpliva nastavitev **Error Trapping**, ki jo dosežemo z ukazom **Options**... iz glavnega menuja **Tools**. V prikazanem pogovornem oknu izberemo jeziček **General**. Pri nastavitvi Error Trapping ima na voljo tri možnosti, ki so opisane v spodnji tabeli:

Način prestrezanja napak	Opis
Break on All Errors	Izvajalno okolje sporoči vsako napako, ne glede na morebitne vzpostavljene prestrezovalnike napak.
Break in Class Module	Izvajalno okolje sporoči napake, ki niso prestrežene v razrednem modulu in se pokažejo v njenem klicatelju. Morebitni klicateljev prestrezovalnik napake ne vpliva na vedenje izvajalnega okolja.
Break on Unhandled Errors	Izvajalno okolje sporoči samo tiste napake, ki niso ujete v nobenem prestrezovalniku, ampak po klicnem skladu »izplavajo« vse do vrha.

Nastavitev različnega načina prestrezanja napak izravna razliko, ki se pokaže pri uporabi posebnih podprogramov iz razrednega modula. Ker jih ne kličemo neposredno, ampak se implicitno prikličejo ob vzpostavitvi in uničenju predmeta ter pri doseganju njegovih lastnosti, je v klicnem skladu vedno vsaj en podprogram, v katerem lahko vzpostavimo prestrezovalnik izvajalne napake.

Slika 17-6: Nastavitev prestrezanja napak



NAMIG Pri nastavitvi prestrezanja, ki sporoči neprestrežene napake v razrednem modulu, lahko posredujemo izvajalno napako naprej po klicnem skladu z izbiro ukaznega gumba Debug, nato pa v kodnem oknu pritisnemo kombinacijo Alt+F5.

Nastavitev, ki jo določimo v pogovornem oknu ukaza Options, postane privzeta za vse nadaljnje projekte. Če želimo določiti način prestrezanja izvajalnih napak samo za trenutni projekt, v priročnem menuju okna s kodo izberemo podmenu **Toggle** in vklopimo želeni način prestrezanja.

Podprogrami lastnosti

V razrednih modulih visual basica lahko zapišemo tri vrste podprogramov za doseg lastnosti. To so:

Podprogram	Opis
Property Get	Funkcija, ki vrača vrednost lastnosti.
Property Let	Procedura, ki nastavi vrednost običajne lastnosti.
Property Set	Procedura, ki nastavi vrednost predmetne lastnosti (torej lastnosti, katere vrednost hrani predmetna spremenljivka).

Razlog za dve različni vrsti nastavitvenih procedur je v posebni skladnji za prirejanje nove vrednosti predmetnih spremenljivk s stavkom Set. Podprogram Property Set se implicitno prikliče pri prirejanju nove vrednosti skozi stavek Set, podprogram Property Let pa v vseh drugih primerih.

NAMIG Poimenovanje procedure Property Let izvira iz davnih dni basica, ko je bilo treba prirejati novo vrednost spremenljivk s

posebnim stavkom **Let**. To skladnjo že ves čas podpira tudi visual basic, le da zapis stavka Let ni več obvezen.

Lastnosti, ki so dostopne za branje in nastavljanje, določimo z dvema podprogramoma, izjema so le lastnosti podatkovnega tipa Variant, ki lahko hrani vse podatkovne tipe, tudi predmete. V tem primeru moramo pri branju vrednosti preveriti vrsto shranjene vsebine, za nastavljanje nove vrednosti pa moramo zapisati proceduri Property Let *in* Property Set. Oglejmo si zgled:

```
' Zasebna vrednost tipa Variant
Private mvntKarkoli

' Branje vrednosti
Public Property Get Karkoli()
  If IsObject(mvntKarkoli) Then
    Set Karkoli = mvntKarkoli
  Else
    Karkoli = mvntKarkoli
  End If
End Property

' Nastavitev vrednosti
Public Property Let Karkoli(ByVal val)
  mvntKarkoli = val
End Property

' Nastavitev predmetne reference
Public Property Set Karkoli(ByVal obj)
  Set mvntKarkoli = obj
End Property
```

Če želimo lastnost izpostaviti samo za branje, preprosto izpustimo proceduri Property Let ali Property Set. Če bi izpustili proceduro Property Get, bi lahko razvili lastnost, dostopno samo za pisanje. Takšne lastnosti so le omejeno uporabne, pogostejša pa je lastnost, v katero lahko vpišemo samo začetno vrednost, zatem pa je dostopna le za branje. Taka je tudi naslednja lastnost, ki določa lastnika predmeta:

```
Private mobjLastnik As Object

Public Property Get lastnik() As Object
  Set lastnik = mobjLastnik
End Property

Public Property Let lastnik(objLastnik As Object)
  If mobjLastnik Is Nothing Then
    mobjLastnik = objLastnik
  Else
    Err.Raise Number:=vbObjectError + 2001, _
      Description:="Lastnost je dostopna le za branje"
  End If
End Property
```

Iz zgledov smo že lahko ugotovili, da je podprogram Property Get pravzaprav funkcija, ki vrača enako vrednost enakega podatkovnega tipa kakor pripada zasebni spremenljivki, ki to vrednost hrani. Po drugi strani sta podprograma Property Let/Set proceduri, ki prejmeta parameter, katerega podatkovni tip ustreza zasebni spremenljivki. Na splošno lahko vsi trije podprogrami prejmejo dodatne parametre. V tem primeru moramo biti pozorni na njihov vrsti red. Pravilo določanja vrstnega reda bi lahko simbolično opisali takole:

```
Dim v As tip

lastnost(p1, p2, … pn) = v ➪ Property Let (p1, p2, … pn, v As tip)
v = lastnost(p1, p2, … pn) ➪ Property Get (p1, p2, … pn) As tip

pi = imei As tipi
```

Parametri podprogramskih lastnosti se morajo ujemati v vrstnem redu, *imenu* in podatkovnem tipu. Dodatne parametre najpogosteje uporabimo za dostop do lastnosti, katerih vrednosti so shranjene v polju (*array property*).

Videli smo, da nam uporaba podprogramskih lastnosti omogoča mnogo večjo prilagodljivost in boljše izkoriščanje prednosti predmetnega programiranja kakor uporaba javnih spremenljivk. Nekaj več dela je s pisanjem kode, vendar nam tudi tu lahko na pomoč priskoči poseben pripomoček. Na splošno velja, da lastnost izpostavimo skozi javno spremenljivko *samo* takrat, ko ne omejujemo dostopa do njene vsebine in že njen podatkovni tip določa nabor sprejemljivih vrednosti. Taki podatkovni tipi so lahko:

- Tip Boolean, ki sprejme le resnično ali neresnično vrednost.
- Realna števila tipov Single in Double, kjer dopuščamo poljubno vrednost.
- Neomejeni nizi s poljubno vsebino.

> **NAMIG** Kdor misli, da bo z rabo javnih spremenljivk napisal učinkovitejšo kodo, ker pri uporabi lastnosti ne bo prišlo do klicev posebnih podprogramov, se moti. Visual basic vsako javno spremenljivko *vedno* ovije z nevidnima podprogramoma zaradi zahtev razredne knjižnice (*type library*).

Metode

Zapis metod se v ničemer ne razlikuje od zapisa kateregakoli javnega podprograma. Metode lahko zapišemo v obliki funkcij, ki vračajo vrednost, ali procedur, ki zgolj opravijo delo. Vrnimo se k našemu razredu za obravnavanje bančnega računa. Dopolnimo ga s kodo, ki omogoča spremljanje stanja na računu ter dvig in polog.

```
' Posodobitev kode
Private mcurStanje As Currency
Private mcurLimit As Currency
```

```
Private Sub Class_Initialize()
  ' določimo začetno vrsto računa
  mvrVrsta = vrVpogled
  mcurStanje = 0
End Sub

' Nova koda
Public Function Stanje() As Currency
  Stanje = mcurStanje
End Function

Public Sub Limit(curVrednost As Currency)
  If mvrVrsta = vrTekoci Then
    mcurLimit = curVrednost
  Else
    Err.Raise _
      Description:="Neveljavna vrsta računa", _
      Number:=vbObjectError + 1001
  End If
End Sub

Public Sub Kredit(curVrednost As Currency)
  If mvrVrsta = vrKredit Then
    mcurLimit = curVrednost
  Else
    Err.Raise _
      Description:="Neveljavna vrsta računa", _
      Number:=vbObjectError + 1001
  End If
End Sub

Public Sub Polog(curVrednost As Currency)
  mcurStanje = mcurStanje + curVrednost
End Sub

Public Sub Dvig(curVrednost As Currency)
  Select Case mvrVrsta
    Case vrVpogled
      If mcurStanje >= curVrednost Then
        mcurStanje = mcurStanje - curVrednost
      Else
        Err.Raise _
          Description:="Prekoračitev računa", _
          Number:=vbObjectError + 1003
      End If

    Case vrTekoci, vrKredit
      If (mcurStanje + mcurLimit) >= curVrednost Then
        mcurStanje = mcurStanje - curVrednost
      Else
        Err.Raise _
          Description:="Prekoračitev računa", _
          Number:=vbObjectError + 1003
      End If
  End Select
End Sub
```

Nič posebnega, izdelali smo le malo bolj trdoživo obliko razreda. Pri uporabi metod se predvsem zastavlja vprašanje, kdaj določeno značilnost razreda izpostaviti prek metode in kdaj z lastnostjo. Splošno pravilo pravi, da so lastnosti stanje predmeta, metode pa njegovo vedenje. Nekatere stvari, kot so barva, oblika, dolžina, so nedvomno lastnosti, vzpostavitev, premik, predstavitev pa nedvomno metode. Seveda pa je tudi mnogo značilnosti, pri katerih je opredelitev nejasna. V zgornjem zgledu smo stanje na računu izpostavili kar prek metode, ki ga vrača, lahko pa bi uporabili tudi lastnost.

Pravzaprav pravo vprašanje ni lastnost ali metoda, vprašati se moramo, kako si želimo določeno značilnost predstavljati. To nas pripelje do pojma *abstrakcija podatkov* (*data abstraction*), ki je prva stopnja pri predmetni analizi. Uporaba predmetnega programiranja nam omogoča, da podatkovne strukture in programsko kodo preoblikujemo v razumljivejšo obliko, po zgledu predmetov iz sveta, ki nas obkroža. Pri bančnem računu smo se pač držali pravila, da za stanje ponavadi vprašamo, zato smo ga predstavili kot metodo. Drugo vprašanje, ki je prav tako domena abstrakcije podatkov, je izbira proceduralne ali funkcijske metode. Metodi Dvig in Polog bi lahko bili tudi funkciji, ki vračata novo stanje.

Nadgradnja razreda

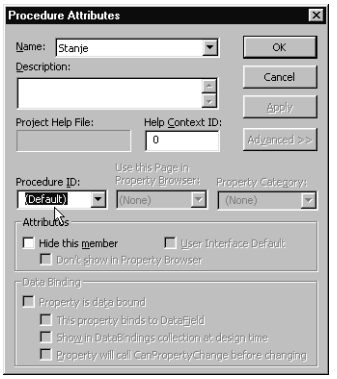
Z določitvijo metod in lastnosti še nismo izčrpali vseh možnosti pri določanju lastnih razredov. Njihovo rabo nam lahko znatno olajša določitev privzetega člana in vpeljava dogodkov (*events*).

Privzeti član

Izbrano lastnost ali metodo razreda lahko določimo za *privzetega člana* razreda (*default member*). Najbolje je za privzetega člana izbrati tisto lastnost ali metodo, ki jo uporabljamo najpogosteje. S tem skrajšamo zapis programske kode in za drobec izboljšamo njeno učinkovitost. Privzeti člani nam še posebej pomagajo pri razvejenih predmetnih modelih, kot je DAO (*Data Access Objects*), ki smo ga spoznali v prejšnjem delu te knjige.

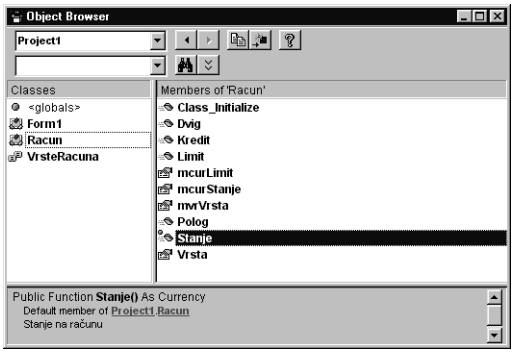
Privzetega člana določimo z ukazom **Procedure Attributes**... iz glavnega menuja **Tools**. V prikazanem pogovornem oknu znotraj seznama **Name** izberemo lastnost ali metodo, ki jo želimo določiti za privzeto. Naj bo to za naš razred metoda »Stanje«. Sedaj kliknemo gumb Advanced, da odpremo preostanek pogovornega okna. S seznama **Procedure ID** izberemo nastavitev »(Default)« in jo potrdimo s klikom gumba OK.

V vsakem razrednem modulu imamo lahko le enega privzetega člana. Če ga želimo spremeniti, moramo najprej izključiti obstoječega. To storimo z zelo podobnim postopkom, le da s seznama Procedure ID izberemo »(None)«.



Slika 17-7: Določanje privzete značilnosti

Isto pogovorno okno nam omogoča, da vsakega člana opremimo s krajšim opisom. Ta opis vnesemo v vnosno polje **Description**. Opisi bodo vidni v predmetnem brskalniku. Gumb Apply nam omogoča, da brez zapiranja okna spremenimo opis in druge atribute za več članov razreda hkrati.



Slika 17-8: Razred Racun v predmetnem brskalniku. Označena je privzeta metoda.

Če smo za privzetega člana določili metodo stanje, lahko obrazec v našem projektu opremimo še s štirimi ukaznimi gumbi (cmdLimit, cmdKredit, cmdPolog, cmdDvig) in oznako (Label1) ter dopolnimo programsko kodo za preizkus razreda Racun:

```
Private r As Racun

' Vzpostavimo modulsko dostopen izvod
Private Sub Form_Load()
  Set r = New Racun
End Sub

Private Sub cmdLimit_Click(Index As Integer)
  r.Limit Text1.Text
End Sub

Private Sub cmdKredit_Click(Index As Integer)
  r.Kredit Text1.Text
End Sub
```

```
Private Sub cmdDvig_Click()
  r.Dvig Text1.Text
  Label1.Caption = r
End Sub

Private Sub cmdPolog_Click()
  r.Polog Text1.Text
  Label1.Caption = r
End Sub

Private Sub Command1_Click()
  r.Vrsta = Text1.Text
End Sub

Private Sub Text1_KeyPress(KeyAscii As Integer)
  If Not (KeyAscii >= Asc("0") And KeyAscii <= Asc("9") _
          Or KeyAscii = 8 Or KeyAscii = 127) Then
    KeyAscii = 0
  End If
End Sub
```

Vidimo, da lahko do privzetega člana pridemo samo z zapisom imena predmeta. Zgledu smo dodali še precej uporabno odzivno proceduro, ki prepreči vnos besedila v vnosno polje.

Dogodki

Doslej smo določali samo *vhodni vmesnik* lastnega razreda. Vhodni vmesnik (*incoming interface*) določa lastnosti in metode, s katerimi lahko od zunaj nadziramo posamezne predmete. Že ves čas pa s pridom izkoriščamo tudi izhodni vmesnik pripravljenih razredov – gradnikov uporabniškega vmesnika. Izhodni vmesnik omogoča vpliv predmeta na zunanje okolje. Uresničimo ga z *dogodki*.

Denimo, da bi naš razred Racun radi dopolnili z dogodkom, ki obvesti zunanji svet o nevarnem znižanju stanja na računu. Nov dogodek napovemo med splošnimi napovedmi razrednega modula s stavkom **Event**:

```
Public Event Kriza()
```

S tem smo napovedali, da je naš razred sposoben sprožiti dogodek (*event*) z imenom Kriza. Vsi dogodki so po privzeti nastavitvi javni, vendar zaradi boljše berljivosti napoved ponavadi opremimo tudi s pridevnikom Public. Preostane nam še, da v programski kodi ob izpolnjenih pogojih dogodek tudi dejansko sprožimo. Dodajmo najprej zasebno konstanto, ki bo določala krizno stanje na računu:

```
Private Const mcOpozorilo = 100
```

Dogodek bomo sprožili ob dvigu:

```
Public Sub Dvig(curVrednost As Currency)
  ' ...
  If mcurStanje < mcOpozorilo Then RaiseEvent Kriza
End Sub
```

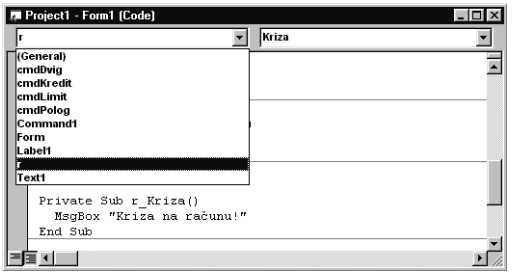
Razred Racun je sedaj postal *vir* dogodkov (*event source*). Prožimo jih s stavkom **RaiseEvent**. Vsi zunanji moduli, ki jih dogodki v razredu zanimajo, se morajo nanje posebej »naročiti«. Naročilo na dogodke dosežemo z napovedjo modulske predmetne spremenljivke z določilom **WithEvents**. V kodi obrazca sedaj spremenimo napoved predmeta r:

```
Private WithEvents r As Racun
```

S tem je obrazec sposoben prisluhniti dogodkom, proženim v razredu Racun. Neposredna posledica te napovedi je sprememba seznamov na vrhu okna s kodo. Na levem seznamu elementov lahko sedaj poleg gradnikov na vmesniku izberemo tudi predmet r. Edini dogodek, ki ga pozna, je zaenkrat dogodek Kriza. Zapišimo še njegovo odzivno proceduro:

```
Private Sub r_Kriza()
  MsgBox "Kriza na računu!"
End Sub
```

Slika 17-9: Nov predmet z lastnim dogodkom



Kadarkoli se bo po dvigu stanje na računu spustilo pod 100, se bo prikazalo sporočilo kot odziv na dogodek Kriza.

Mehanizem dogodkov je pri lastnih razredih popolnoma enak kakor pri gradnikih uporabniškega vmesnika. Edina razlika izvira iz narave lastnih razredov. Medtem ko so gradniki vzpostavljeni samodejno, ob vzpostavitvi obrazca, je treba predmete lastnih razredov ustvariti ročno. Od stvaritve pa do njihovega konca so dostopni tudi dogodki. Vzpostavitev predmetov z dogodki je ponavadi najbolje izdelati znotraj odziva na dogodek Initialize ali Load, če gre za obrazec, tako kakor v zgornjem zgledu.

Seveda je mogoče dogodke uporabljati tudi v drugih razrednih modulih, celo v istem modulu, v katerem so napovedani. V tem primeru je včasih koristno uporabiti tudi zasebne dogodke, ki rabijo le za komuniciranje znotraj modula.

NAMIG Razvito programsko kodo najdete na priloženi plošči CD-ROM v mapi \Primeri\4-Razredi\1-Lastni razredi\3-Nadgradnja\.

Parametri

Omenili smo, da je mehanizem dogodkov pri gradnikih in pri lastnih razredih enak. Kakor vemo, so dogodki, ki jih spremljajo dodatni parametri, in dogodki, pri katerih je z določitvijo nove vrednosti parametra mogoče vplivati na nadaljnji potek izvajanja kode (npr. dogodek QueryUnload). Take dogodke lahko razvijemo tudi v lastnih razrednih modulih.

Popolna oblika napovedi dogodka, zapisana v simbolični obliki, se glasi:

```
[[Public|Private]] Event dogodek ([seznam parametrov])
```

Seznam parametrov je zelo podoben seznamu parametrov podprograma z nekaj izjemami:

- Dogodki ne morejo imeti izbirnih ali poimenovanih parametrov.
- Dogodki ne morejo imeti nedoločljivega števila parametrov.

Oglejmo si demonstracijo dogodka, ki nam omogoča dvosmerno komuniciranje med virom in poslušalcem. Posnemali bomo dolgotrajnejše opravilo, ki vsako sekundo sporoči napredek poslušalcem. Hkrati jim ponudi tudi možnost, da nadaljevanje izvajanja opravila prekinejo. Napovedati moramo naslednji dogodek:

```
Public Event Obdelano(ByVal odstotek As Single, ByRef preklic As Boolean)
```

Dogodek ima dva parametra. Prvi bo navzven sporočal odstotek opravljenega dela, drugi pa se bo prenašal po referenci in bo poslušalcu dogodka omogočal preklic izvajanja.

Sprogramirati moramo javno metodo, ki posnema opravilo in proži zgoraj napovedani dogodek:

```
Public Sub DolgotrajnoOpravilo(ByVal intRajanje As Integer)
Dim lngStart As Long
Dim intSekunde As Integer
Dim intProzi As Integer
Dim blnPreklic As Boolean

  blnPreklic = False
  intProzi = 0
  lngStart = Int(Timer)

  Do

    DoEvents
    intSekunde = Int(Timer - lngStart)
    If intSekunde > intProzi Then
      intProzi = intSekunde
      RaiseEvent Obdelano(intSekunde / intRajanje, blnPreklic)
      If blnPreklic Then Exit Sub
    End If
  Loop Until intSekunde >= intRajanje

End Sub
```

Kakor vidimo, se zanka prekine tudi, ko je posredovani parameter preklic dobil resnično vrednost.

Pri poslušalcu bi bila lahko programska koda naslednja:

```
Private WithEvents d As Dogodki

Private Sub Form_Load()
  Set d = New Dogodki
End Sub

Private Sub Command1_Click()
  d.DolgotrajnoOpravilo 10
End Sub

Private Sub d_Obdelano(ByVal odstotek As Single, preklic As Boolean)
  Command1.Caption = odstotek
  If odstotek > 0.3 Then preklic = True
End Sub
```

Zaradi poslušanja dogodkov predmetne spremenljivke ne moremo napovedati z določilom New, zato nov izvod ustvarimo v odzivu na dogodek Load samega obrazca. Po kliku ukaznega gumba sprožimo deset sekund dolgo opravilo. Vsako sekundo se bo pripetil dogodek »Obdelano«, ki ga po tridesetih odstotkih opravljenega dela – v našem primeru je to po treh sekundah – prekinemo.

Večličnost

Če več razredov ponuja podoben vmesnik, lahko presek njihovih skupnih metod ali lastnosti uporabimo na predmetni spremenljivki, ne da bi natančno vedeli, kateremu razredu pripada. Tako lahko npr. uporabimo predmete, zbrane v zbirki Controls. Za vse vemo, da so gradniki, ne vemo pa vnaprej, kateremu razredu pripadajo. Ker imajo vsi gradniki zagotovo lastnost Name, lahko zapišemo tak sprehod prek zbirke:

```
Dim cnt As Control
For Each cnt In Controls
  Debug.Print cnt.Name
Next
```

Tej značilnosti uporabe splošnih predmetnih spremenljivk pravimo *večličnost*. Večličnost (*polymorphism*) je v večini predmetno usmerjenih razvojnih orodij posledica mehanizma *dedovanja* (*inheritance*). Visual basic dedovanja ne pozna, zato omogoča večličnost skozi izpeljavo več vmesnikov.

> **NAMIG** Posledica uporabe vmesnikov je zelo hitro dinamično razreševanje naslova ustrezne metode, saj je vsak specifičen razred neposredna nadgradnja enega ali več vmesnikov.

Abstraktni razred

Večkrat smo že omenili, da je vmesnik skupek lastnosti in metod, s katerimi je mogoče vplivati na stanje in vedenje predmeta. Če želimo izkoristiti večličnost, moramo določiti vmesnik, ki ga bo implementiralo več razredov. Definicijo vmesnika določimo z *abstraktnim* razredom. Abstraktni razred (*abstract class*) je razred z napovedmi metod in lastnosti, vendar so njegove posamezne metode lahko brez pripadajoče programske kode. S tem določimo predlogo stanja in vedenja, iz katere izvirajo sorodni razredi.

> **NAMIG** V predmetnem izrazoslovju s pridevnikom *abstraktni* označimo tiste razrede, iz katerih ni mogoče izdelati predmetov, temveč rabijo le kot predloga drugim razredom. Po tem merilu v visual basicu ni mogoče izdelati pravih abstraktnih razredov.

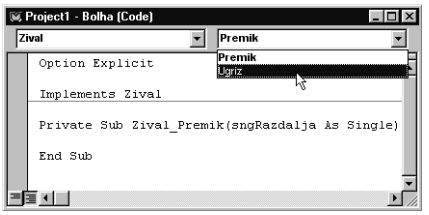
Za zgled bomo razvili nov projekt, ki bo demonstriral uporabnost večličnosti in uporabe vmesnikov. Odpremo nov projekt vrste »Standard EXE«. Vanj bomo dodali nov razredni modul, v katerem bomo določili predlogo razredov živali. Poimenujmo ga »Zival« in mu dodajmo naslednjo kodo:

```
Public Sub Premik(sngRazdalja As Single)

End Sub

Public Sub Ugriz(objPredmet As Object)

End Sub
```

Razred Zival je abstraktni razred, ki ne obsega programske kode, temveč le napoved vmesnika. Z njim bomo uresničili večlično vedenje. Sedaj dodajmo nov razredni modul. Poimenujmo ga »Bolha«, v okno s kodo pa zapišimo naslednjo vrstico kode:

```
Implements Zival
```

S stavkom **Implements** smo se zavezali, da bomo v nadaljevanju poskrbeli za izpeljavo *celotnega* vmesnika, določenega v abstraktnem razredu Zival. Ne moremo izbrati le dela vmesnika, vedno moramo zapisati kodo za *vse* metode in lastnosti. Posledica zgornjega stavka je spremenjena vsebina seznamov v oknu s kodo. Znotraj seznama elementov, levo zgoraj, je sedaj mogoče izbrati razred Zival, desni seznam pa omogoča dostop do metod Premik in Ugriz.



Slika 17-10: Implementacija vmesnika

Izberimo metodi in jima dodajmo naslednjo programsko kodo:

```
Private Sub Zival_Premik(sngRazdalja As Single)
  Debug.Print "Bolha se je premaknila: " & sngRazdalja
End Sub

Private Sub Zival_Ugriz(objPredmet As Object)
  Debug.Print "Bolha je ugriznila " & TypeName(objPredmet)
End Sub
```

Mimogrede smo spoznali novo funkcijo. Funkcija **TypeName**, ki jo uporabljamo pri izpeljavi metode Ugriz, vrne opisno ime razreda. Zakaj sta proceduri napovedani kot zasebni? Zato ker sta dostopni prek vmesnika abstraktnega razreda Zival. Če bi proceduri napovedali kot javni, bi bili del javnega vmesnika razreda Bolha. Da, tudi razred Bolha ima svoj vmesnik, ki pa je zaenkrat brez članov, saj mu nismo dodali nobene nove javne lastnosti ali metode.

Vmesnik Zival bomo izpeljali v še enem razrednem modulu, Dinozaver:

```
' Razredni modul Dinozaver
Implements Zival

Private Sub Zival_Premik(sngRazdalja As Single)
  Debug.Print "dinozaver se je premaknil: " & sngRazdalja
End Sub

Private Sub Zival_Ugriz(objPredmet As Object)
  Debug.Print "Dinozaver je ugriznil " & TypeName(objPredmet)
End Sub
```

Razred Zival nam omogoča večličnost, saj sta njegov vmesnik izpeljala dva nova razreda. Za prikaz uporabe se premaknimo na obrazec, ki mu dodamo ukazni gumb. Gumbu priredimo naslednjo odzivno proceduro:

```
Private Sub Command1_Click()
Dim blh As Bolha
Dim dno As Dinozaver
Dim zvl As Zival

  Set blh = New Bolha
  Set dno = New Dinozaver

  Set zvl = blh
  zvl.Ugriz dno

  Set zvl = dno
  zvl.Ugriz blh

End Sub
```

Po zagonu programa lahko v takojšnjem oknu preberemo:

```
Bolha je ugriznila Dinozaver
Dinozaver je ugriznil Bolha
```

Sprehodimo se po programski kodi korak za korakom (ukaz Step Into ali tipka F8). Čeprav v obeh primerih uporabljamo predmetno spremenljivko zvl, katere tip je

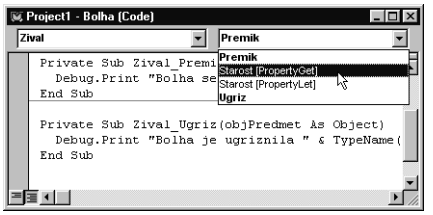
razred Zival, se enkrat pokliče ugriz v razredu Bolha, drugič pa ugriz v razredu Dinozaver. Predmetna spremenljivka zvl lahko kaže na katerikoli predmet razreda, ki izpelje vmesnik Zival.

Izpeljava lastnosti

Poglejmo, kaj se zgodi, če abstraktnemu razredu dodamo lastnost, izpostavljeno skozi javno spremenljivko. Naj bo to starost živali. V del javnih napovedi razreda Zival zapišemo:

```
' Razredni modul Zival
Public Starost As Date
```

V kodnih razredih Bolha in Dinozaver bo seznam članov razreda Zival sedaj dopolnjen s *podprogramsko* lastnostjo Starost, kar prikazuje naslednja slika:



Slika 17-11: Podprogramska lastnost, ki zahteva izpeljavo

Zgornja slika nam nazorno ponazarja dejstvo, da je lastnost, izpostavljena kot javna spremenljivka, zgolj priročni zapis podprogramske lastnosti. Visual basic jo vedno pretvori v nekaj podprogramov Property.

Razred mora izpeljati vse elemente vmesnika, zato bomo razredoma Bolha in Dinozaver dodali zasebno spremenljivko, ki bo hranila starost živali, in par procedur, ki bosta nastavljali oziroma vračali to vrednosti:

```
Private mdteStarost As Date

Private Property Let Zival_Starost(ByVal RHS As Date)
  mdteStarost = RHS
End Property

Private Property Get Zival_Starost() As Date
  Zival_Starost = mdteStarost
End Property
```

Zasebno spremenljivko moramo dodati ročno, saj gre za podrobnost izpeljave abstraktnega razreda. Visual basic ne zna ugotoviti imena zadnjega parametra v proceduri Property Let, zato ga poimenuje RHS. V zgornjem zgledu nismo posebej omejevali vrednosti, ki so na voljo za starost posamezne živali, čeprav nam tega nihče ne prepoveduje. Prav enostavno bi lahko omejili starost bolh na največ nekaj mesecev (hm, le koliko je življenjska doba bolh?):

```
Private Property Let Zival_Starost(ByVal RHS As Date)
  If RHS < 100 Then
    mdteStarost = RHS
  Else
    Err.Raise Number:=vbObjectError + 2001, _
      Description:="Neustrezna starost bolhe!"
  End If
End Property
```

No, omejevanje starosti, ki jih lahko priredimo posamezni živali, morda ni najboljši pristop. Žival bi se morala zavedati svoje starosti in jo izpostaviti kot lastnost, dostopno le za branje. V tem primeru moramo posodobiti vmesnik v razredu Zival:

```
' Razredni modul Zival
' nič več: Public Starost As Date
' sedaj samo za branje:
Public Property Get Starost() As Date

End Property
```

V razrednih modulih, ki izpeljujeta ta vmesnik, moramo sedaj poskrbeti za vzpostavitev starosti in posodobiti funkcijo Property Get, proceduro Property Let pa odstranimo:

```
Private mdteRojena As Date

Private Sub Class_Initialize()
  mdteRojena = Now
End Sub

Private Property Get Zival_Starost() As Date
  Zival_Starost = Now - mdteRojena
End Property
```

Različni vmesniki

Kot smo ugotovili med preizkušanjem razredov živali, ni posebne razlike med »abstraktnim« razredom in razredi, ki izpeljujejo njegov vmesnik. V visual basicu so vsi preprosto razredi. Vsakemu razredu pripada *privzeti* vmesnik (*default interface*), ki ima enako ime kakor sam razred, z eno samo razliko: ime privzetega vmesnika se začne z znakom podčrtaj (_). Ta znak označuje, da je vmesnik skrit v predmetni knjižnici.

Zato ima npr. razred Bolha privzeti vmesnik, imenovan _Bolha. Ker izpeljuje vmesnik razreda Zival, mu pripada tudi ta vmesnik. Torej razred Bolha izpeljuje dva vmesnika. Seveda, lahko bi jih še več. No, vsak razred, izpeljan iz razreda Bolha, je vedno bolha. To ponazarja naslednji zgled:

```
Dim blh As Bolha
Dim zvl As Zival

  Set blh = New Bolha
  Set zvl = blh
  MsgBox TypeName(zvl) ' Bolha
```

Prikaže se sporočilo z vsebino »Bolha« in ne »Zival«. Kadarkoli priredimo spremenljivki zvl predmetno spremenljivko drugega razreda – v našem zgledu razreda Bolha –, ga visual basic povpraša, ali podpira vmesnik razreda Zival. Če bi bil odgovor nikalen, bi se sprožila izvajalna napaka.

Kaj pa se zgodi pri uporabi generičnih razredov, kot je npr. Object? Dodajmo dinozavrom novega člana njihovega vmesnika, ki je zaenkrat prazen:

```
' Razredni modul Dinozaver
'  ...
Public Sub Zarjovi()
  Debug.Print "Grrrrr"
End Sub
```

Poizkusimo sedaj z naslednjim zgledom:

```
Dim dno As Dinozaver
Dim zvl As Zival
Dim obj As Object

  Set dno = New Dinozaver
  Set zvl = dno
  Set obj = zvl
  MsgBox TypeName(obj) ' Dinozaver
  obj.Premik (10)
  obj.Zarjovi          ' Napaka, nepodprta metoda
```

Čeprav se predmet še vedno predstavlja kot Dinozaver, ne podpira več njegovega vmesnika. Pri uporabi generičnega razreda Object se moramo zavedati, da je podprti vmesnik vedno vmesnik razreda, ki smo ga spremenljivki priredili nazadnje. V zgornjem zgledu je to vmesnik razreda Zival, ki metode Zarjovi ne pozna. V splošnem generičnega razreda niti ne uporabljamo, saj je veliko počasnejši pri *povezovanju* članov. Načine povezovanja bomo spoznali v nadaljevanju.

Pri izpeljevanju vmesnikov nimamo na voljo le lastnih razredov. Izpeljemo lahko razred iz poljubne predmetne knjižnice, ki podpira avtomatizacijo (*automation*). Predmetne knjižnice so naštete na seznamu, ki ga prikaže ukaz **References**... iz glavnega menuja **Project**. Prava moč predmetnih knjižnic se pokaže, ko z *zadolžitvijo* izkoriščamo že pripravljeno kodo.

Zadolžitve

Ugotovili smo, da znotraj visual basica ni posebne razdelitve na abstraktne in neabstraktne razrede. Iz vsakega razreda lahko izdelamo predmet in vsak razred lahko izpelje vmesnik drugega razreda. To nas pripelje do morda največje prednosti predmetnega programiranja: *vnovične izrabe* že napisane kode (*code reuse*). V visual basicu jo dosežemo z izpeljavo funkcionalnega razreda.

Znotraj razreda, ki izpeljuje vmesnik drugega razreda, lahko vzpostavimo zasebno spremenljivko, ki se sklicuje na ta drugi razred. S tem dobimo notranji predmet (*inner object*), ki nam je na voljo s svojo funkcionalnostjo. Pri izpeljavi abstraktnih razredov smo omenili, da je treba zapisati kodo, ki obravnava vse člane vmesnika. Z notranjim predmetom lahko izbiramo, katera opravila bomo enostavno posredovali njemu, katera bomo malce spremenili in katera popolnoma predrugačili. Posredovanje opravil notranjemu predmetu imenujemo tudi *zadolžitev* (*delegation*).

Za zgled bomo abstraktni razred živali dopolnili z vsem potrebnim za spremljanje starosti posameznega primerka. Saj se koda pri bolhi in dinozavru tako ali tako precej podvaja. Razred Zival bomo dopolnili:

```
' Razredni modul Zival
Private mdteRojena As Date

Private Sub Class_Initialize()
  mdteRojena = Now
End Sub

Public Property Get Starost() As Date
  Starost = Now - mdteRojena
End Property
```

Seveda, naj bo razred Zival kar največji skupni imenovalec vseh zivali. Starost zagotovo pripada vsaki. Znotraj razrednih modulov bolhe in dinozavra bomo popravili kodo, ki bo sedaj izkoriščala notranji predmet razreda Zival:

```
' Razredni modul Dinozaver
' Notranji predmet
Private mZival As Zival

Private Sub Class_Initialize()
  ' Vzpostavitev notranjega predmeta
  Set mZival = New Zival
End Sub

Private Property Get Zival_Starost() As Date
  ' Zadolžitev
  Zival_Starost = mZival.Starost
End Property
```

Vnovična uporaba že pripravljene kode je silno močno programersko orodje z zelo veliko prilagodljivostjo. Ker je zadolžitev zgolj uporaba vmesnika notranjega predmeta, nam nihče ne prepoveduje, da bi pred zadolžitvijo ali po njej vpletli še kaj lastne kode. Bolha bi lahko npr. lagala o svojih letih:

```
Private Property Get Zival_Starost() As Date
  ' Zadolžitev
  Zival_Starost = mZival.Starost / 2
End Property
```

NAMIG Za boljše razumevanje uporabe notranjih predmetov in zadolžitev je najuporabnejše orodje koračno izvajanje. Vsaj enkrat se sprehodimo po zgledu rabe z ukazom Step Into (oz. s tipko F8).

Programsko kodo zgleda večličnosti najdete na priloženi plošči CD-ROM v mapi \Primeri\4-Razredi\1-Lastni razredi\4-Veclicnost\.

Povezovanje

Iz zgledov smo videli, da predmetna spremenljivka lahko kaže na generični predmet razreda Object, na bolj specifičen abstraktni razred, npr. Zival, ali na dokončno izdelan razred, npr. Bolha. Razred, ki ga priredimo predmetni spremenljivki, določa njeno *povezovanje*. Povezovanje (*binding*) določa način, kako visual basic ugotovi, katere metode in lastnosti (kakšen vmesnik) izbrana predmetna spremenljivka sploh podpira. Povezovanje močno vpliva na trdoživost in učinkovitost programske kode. Poznamo dve glavni vrsti povezovanja: pozno in zgodnje, ki se deli še na dve vrsti. Oglejmo si njihove značilnosti.

Pozno povezovanje

Pozno povezovanje (*late binding*) se uporabi pri predmetnih spremenljivkah, katerih tip je generičen. Generični razredi so Object, Variant, pri delu z gradniki vmesnika pa tudi Form in Control. Pri teh predmetih visual basic med prevajanjem kode ne more ugotoviti, kakšen vmesnik podpirajo, zato ob vsakem klicu metode ali posegu po lastnosti doda nekaj kode, ki bo preverila vmesnik predmeta med izvajanjem. Zato je učinkovitost kode precej manjša. Vsaka uporaba predmeta je počasnejša za okrog 50 %. Hkrati se odpira možnost, da pride do napak v kodi, saj lahko uporabimo napačno metodo ali lastnost in to ugotovimo šele med izvajanjem.

NAMIG Pozno povezovanje ob *vsaki* uporabi predmeta med izvajanjem programa najprej ugotavlja njegovo enolično oznako (*dispatch ID*), ki jo uporabi za povezovanje z ustreznim vmesnikom. To v praksi pomeni dvakrat daljši čas dostopa do vsake posamezne lastnosti ali metode kakor pri zgodnjem povezovanju.

Zgodnje povezovanje

Če predmetni spremenljivki predpišemo specifičen razred, bo visual basic že med prevajanjem znal ugotoviti, kakšen vmesnik podpira. Uporaba predmeta bo bistveno hitrejša, hkrati pa bo trdoživost kode močno izboljšana, saj bo napačna uporaba razreda večinoma ujeta že pri prevajanju[58]. Uporaba zgodnjega povezovanja

[58] Pri prevajanju se izplača izbrati možnost Start With Full Compile (Ctrl+F5), ki pred zagonom prevede celotno kodo projekta.

je zato najbolj priporočljiv način napovedovanja spremenljivk. Generične tipe uporabljamo samo izjemoma.

NAMIG Vrsto povezovanja povsem določa napoved spremenljivke. Pri zgodnjem povezovanju pa na hitrost dela vpliva tudi razredna knjižnica, ki ji specifični razred pripada.

Uporaba vseh razredov, ki jih razvijemo sami, uporablja najhitrejši način zgodnjega povezovanja z virtualno tabelo naslovov podprogramov (*vtable*). Vsakemu klicu metode ali posegu po lastnosti se med prevajanjem predpiše ustrezen odmik v virtualni tabeli. Med izvajanjem je zato povezovanje izredno hitro, saj zahteva en sam vpogled v to tabelo.

Če predmetna knjižnica, ki ji pripada izbrani specifični razred, ne podpira virtualnih tabel, bo visual basic uporabil počasnejšo obliko zgodnjega povezovanja z oznakami razredov (*dispatch ID*). Ker bo oznako razreda ugotovil že med prevajanjem, je tovrstno povezovanje še vedno bistveno hitrejše od poznega povezovanja.

Predmetni model

Predmete je v visual basicu precej enostavno uporabljati. Predmet postane dostopen z vzpostavitvijo predmetne spremenljivke, ki kaže nanj (njegove reference). Predmet umre, ko nanj ne kaže nobena predmetna spremenljivka več. Nadzor enostavnih programov z majhnim številom dejavnih predmetov je dokaj enostaven. Vsakemu predmetu lahko pripada druga predmetna spremenljivka. Kaj pa, če je predmetov mnogo več? Seveda lahko posežemo po polju predmetnih spremenljivk ali pa vzpostavimo *zbirko* (*collection*) sorodnih predmetov. Njihovo vzdrževanje pa lahko postane precej zapleteno, saj bomo težko ugotavljali, kdaj je določen predmet še dejaven in kdaj je zaradi izgube zadnje reference umrl. Določenih predmetov se bomo zelo težko znebili, saj bomo enostavno pozabili na neko referenco, ki jih še vedno vzdržuje pri življenju.

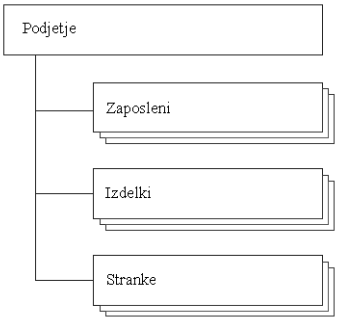
Rešitev je razvoj *predmetnega modela*. Predmetni model (*object model*) nam omogoča, da razrede združimo v hierarhično strukturo. Nadrejenim razredom lahko omogočimo popoln nadzor nad podaniki z lastnimi zbirkami. Za zgled predmetnega modela lahko vzamemo model DAO (*Data Access Objects*), ki smo ga uporabljali v prejšnjem delu te knjige.

NAMIG Nadzor predmetnih referenc je še posebej zapleten, če so predmetne spremenljivke napovedane z implicitno stvaritvijo. Takih predmetov se zelo težko znebimo, saj se ob prvem dostopu do njihove

lastnosti ali metode samodejno ustvarijo znova. Implicitno stvaritev uporabljamo le za kodiranje nekaterih posebnih značilnosti razredov, drugače pa se jih izogibamo. Koristno uporabo implicitne stvaritve bomo spoznali v nadaljevanju, pri razvijanju lastne zbirke.

Zgled modela

Za zgled predmetnega modela bi si lahko ogledali abstrakcijo podjetja. Podjetje ima svoje uslužbence, svoje izdelke in svoje stranke. Vsakega od teh lahko predstavimo z lastnim predmetom. In ker je uslužbencev, izdelkov in strank lahko več, jih bomo združili v zbirko. Zgled predmetnega modela prikazuje naslednja slika:



Slika 17-12: Predmetni model podjetja

Tak model bi lahko pripravili na več načinov:

- Zbirke uslužbencev, izdelkov in strank bi izpostavili kot javne lastnosti razreda Podjetje. To je najhitrejša, vendar površna in ranljiva rešitev.

```
' Razredni modul Podjetje
Public Usluzbenci As Collection
Public Izdelki As Collection
Public Stranke As Collection
```

- Zbirke bi shranili v zasebnih predmetnih spremenljivkah, dostop do njih pa bi omogočili skozi vrsto javnih metod. Ta način je najmanj predmeten.

```
' Razredni modul Podjetje
Private mcolUsluzbenci As Collection
Private mcolIzdelki As Collection
Private mcolStranke As Collection

' sledijo metode za delo z zbirkami
```

- Za vsako zbirko bi izdelali lasten razred. Te zbirke bi izpostavili tudi skozi razred Podjetje kot lastnosti, dostopne samo za branje. To je najbolj trdoživ in najbolj predmeten način.

```
' Razredni modul Podjetje
Private mcolUsluzbenci As New Usluzbenci
' in še druge zbirke…

' Lastnost izpostavimo kot lastnost, dostopno le za branje
Public Property Get Usluzbenci() As Usluzbenci
  Set Usluzbenci = mcolUsluzbenci
End Property
' in še druge lastnosti…

' Določiti je treba razrede Usluzbenec, Usluzbenci;
' Izdelek, Izdelki in Stranka, Stranke
```

Vidimo, da je ključ do rešitve vedno v zbirki, ki je lahko bodisi predmet vgrajenega razreda Collection bodisi posebej izpeljana zbirka za specifične potrebe.

Zbirke

Z zbirkami predmetov (*collections*) se srečujemo že precej časa. Vemo, da združujejo več predmetov istega razreda v pregledno celoto. Posamezen predmet v zbirki lahko dosežemo z njegovim zaporednim številom ali z opisno oznako, zapisano v nizu znakov. Sprehod po vseh elementih zbirke je mogoč z zanko For Each.

Zbirke so nam na voljo tudi kot predmet razreda **Collection**, katerega vmesnik lahko uporabimo v novih razredih in s tem oblikujemo lastne zbirke s podobnimi lastnostmi, kakor jih imajo že pripravljene zbirke, ki smo jih uporabljali doslej (npr. Controls). Predmet razreda Collection rabi podobnemu namenu kakor polje spremenljivk prilagodljivega tipa Variant. Zbirke pa imajo v primerjavi s polji dve zelo učinkoviti prednosti:

- Pri dodajanju ali odvzemanju elementov nam ni treba spreminjati velikosti, tako kakor pri poljih.
- Elemente zbirke lahko dosegamo *asociativno*, z opisno oznako. Pri poljih imamo na voljo le celoštevilčno zaporedno številko.

Vgrajeni razred Collection obsega naslednje člane:

Član	Opis
metoda Add	Doda nov element v zbirko.
metoda Remove	Odstrani element iz zbirke.
metoda Item	Vrne posamezen element. To je tudi privzeti član razreda.
lastnost Count	Vrne število vsebovanih elementov v zbirki.

Natančen opis parametrov posameznih metod najdemo v sprotni pomoči.

Čeprav pri delu z zbirkami najpogosteje uporabljamo opisne oznake, je mogoče do posameznega elementa zbirke priti tudi s celim zaporednim številom. Uporabi zaporednega števila se izogibamo iz več razlogov:

- Istemu elementu se zaporedna številka lahko večkrat spremeni. Ob vsakem dodajanju ali odvzemanju elementov so zaporedne številke na novo določene.
- Prvo zaporedna številka je pri nekaterih zbirkah 0 (*zero-based*), pri drugih pa 1 (*one-based*). Vgrajeni razred Collection uporablja za prvo zaporedno številko enico, kar je običajno tudi pri vseh novejših zbirkah, dodanih v zadnjih dveh različicah visual basica.

Lastna zbirka

Uporabo lastne zbirke si bomo ogledali na zgledu izgradnje najprimernejše oblike predmetnega modela, ki ponazarja podjetje. Odprimo nov, prazen projekt vrste »Standard EXE«, dodajmo mu nov razredni modul in ga poimenujemo Podjetje. Najprej bomo določili nadrejeni razred celotnega predmetnega modela:

```
' Razredni modul Podjetje

Private mcolUsluzbenci As New Usluzbenci
' Private mcolIzdelki As New Izdelki
' Private mcolStranke As New Stranke

Public Ime As String

Public Property Get Usluzbenci()
  Set Usluzbenci = mcolUsluzbenci
End Property
```

Napovedali smo zasebno predmetno spremenljivko, ki nam bo pomagala izdelati lastnosti Usluzbenci, dostopne le za branje. Na voljo imamo dva pristopa:

- Spremenljivko lahko napovemo z implicitno stvaritvijo. Nov izvod razreda Collection se bo samodejno izdelal ob prvi uporabi njegove lastnosti ali metode (ta pristop smo uporabili zgoraj).
- Spremenljivko napovemo kakor navadno ter zapišemo odziv na dogodek Initialize, ki vzpostavi nov izvod razreda Collection.

Prvo ponuja več udobja, z drugim pa zapišemo preglednejšo kodo. Izberemo pač tisto pot, ki bolj ustreza naši lenobi oziroma redoljubju. Če želimo popolnoma izdelan predmetni model, bi morali poskrbeti še za zbirko izdelkov in strank (v komentarju), ki pa sta praktično povsem enaki zbirki Usluzbenci, zato bomo njun razvoj izpustili. Razred ima še eno javno lastnost, ime podjetja.

Preden se lotimo izdelave lastne zbirke Usluzbenci, moramo pripraviti razredni modul, ki bo opisoval posamezen element te zbirke. Odprimo nov razredni modul in ga poimenujmo Usluzbenec (ednina!):

```
' Razredni modul Usluzbenec

' Zaščitena lastnost ID
Private mstrID As String

Public Ime As String
Public Placa As Currency

Public Property Get ID() As String
  ID = mstrID
End Property

' Omogočimo le en zapis lastnosti ID
Public Property Let ID(strID As String)
Static blnZnova As Boolean
  If Not blnZnova Then
    mstrID = strID
    blnZnova = True
  Else
    Err.Raise Number:=vbObjectError + 3001, _
      Description:="Lastnost ID je dostopna le za branje!"
  End If
End Property
```

Razred ni nič posebnega. Poleg podatkovne strukture, ki hrani zapis o enem delavcu, omejuje dostop do lastnosti ID, ki je njegova enolična oznaka. Lastnost smo kodirali tako, da dovoljuje samo enkraten vpis, zatem pa je na voljo le za branje.

Sedaj se lahko posvetimo ključnemu problemu, izdelavi lastne zbirke Usluzbenci. Tudi zanjo odpremo nov razredni modul, ki ga ustrezno poimenujemo. Lastne zbirke izdelamo tako, da zelo izkoriščamo storitve že pripravljenega razreda Collection. Čeprav vmesnika tega razreda ne moremo izpeljati neposredno, s stavkom Implements, pa nam nihče ne prepoveduje, da ne bi *zadolžili* (*delegate*) njegovih metod in lastnosti. Za začetek bomo napovedali zasebno predmetno spremenljivko:

```
' Razredni modul lastne zbirke Usluzbenci

' Zaščitena lastnost
Private mcolZbirka As New Collection
```

Tudi ta zasebna zbirka je določena z implicitno stvaritvijo. Nov izvod zbirke se bo samodejno izdelal ob prvem posegu do njene lastnosti ali metode. Najprej si oglejmo, kako bomo dodali novega uslužbenca:

```
' Metode zadolžijo pripravljeni razred Collection
' - Dodaj vzpostavi nov predmet razreda Usluzbenec
'   in ga doda v zbirko
Public Function Dodaj(ByVal strIme As String, ByVal curPlaca As Currency)
Static intID As Integer
Dim uslNovi As New Usluzbenec
  With uslNovi
    intID = intID + 1
    .ID = "U" & Format(intID, "00000")
```

```
    .Ime = strIme
    .Placa = curPlaca
    mcolZbirka.Add uslNovi, .ID
  End With
  Set Dodaj = uslNovi
End Function
```

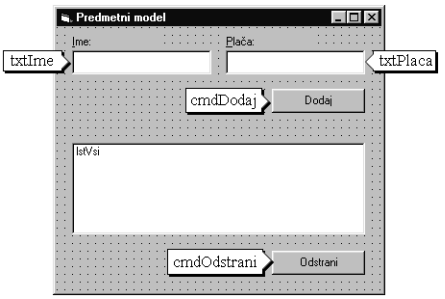
Vse delo pravzaprav opravi metoda Add pripravljenega razreda Collection. Mi samo poskrbimo za izračun enolične oznake delavca. Pri tem nam pomaga statični števec, ki se lahko samo povečuje in med izvajanjem programa nikoli ne bo zasedel dveh enakih vrednosti. Ker metoda Add zahteva enoličen ključ v obliki niza znakov, števec združimo z znakom U, tako da dobimo enolične ključe v obliki »U00001«, »U00002« itn. Ker smo metodo Dodaj zapisali kot funkcijo, na koncu poskrbimo še za posredovanje pravkar dodanega novega delavca.

Druge metode so še preprostejša oblika zadolžitve razreda Collection:

```
' - Preštej uporabi lastnost Count
Public Function Prestej() As Long
  Prestej = mcolZbirka.Count
End Function

' - Odstrani uporabi metodo Remove
Public Function Odstrani(ByVal vntIndeks As Variant)
  mcolZbirka.Remove vntIndeks
End Function

' - Vrni element je preslikava lastnosti Item,
'   določimo jo tudi za privzetega člana razreda
Public Function Element(ByVal vntIndeks As Variant)
  Set Element = mcolZbirka.Item(vntIndeks)
End Function
```

Uporabo novo razvite zbirke lahko olajšamo tako, da metodo Element označimo kot privzeto metodo razreda. Privzeto metodo določimo z ukazom **Procedure Attributes**... iz glavnega menuja **Tools** (postopek smo opisali zgoraj, glej Nadgradnja razreda).

Za zgled uporabe novega predmetnega modela bomo izkoristili obstoječi obrazec v našem projektu. Njegov uporabniški vmesnik oblikujmo po zgledu z naslednje slike:



Slika 17-13: Uporabniški vmesnik za preizkus predmetnega modela

Nastaviti moramo naslednje lastnosti:

Gradnik	Lastnost	Nastavitev
Vnosno polje (*TextBox*)	(Name) Text	txtIme (prazno)
Vnosno polje (*TextBox*)	(Name) Text	txtPlaca (prazno)
Ukazni gumb (*CommandButton*)	(Name) Caption	cmdDodaj Dodaj
Ukazni gumb (*CommandButton*)	(Name) Caption	cmdOdstrani Odstrani
Odprti seznam (*ListBox*)	(Name)	lstVsi

Dodajmo mu še programsko kodo:

```
' Obrazec Form1
Private pod As Podjetje

Private Sub Form_Load()
  Set pod = New Podjetje
  pod.Ime = "Superultramegaok d.o.o."
End Sub

Private Sub cmdDodaj_Click()
  pod.Usluzbenci.Dodaj txtIme.Text, CCur(txtPlaca.Text)
  OsveziSeznam
End Sub
```

Koda uporablja storitve pripravljenih razredov. Določili smo zasebno predmetno spremenljivko, ki bo opisovala podjetje. To spremenljivko najprej vzpostavimo in podjetje tudi poimenujemo. Sedaj lahko z njeno lastnostjo Usluzbenci, ki je pravzaprav zbirka, dodajamo uslužbence. Po vsakem dodajanju pokličemo lastno proceduro, ki poskrbi za obnovitev vsebine seznama. Kako se glasi ta procedura?

```
Private Sub OsveziSeznam()
Dim i As Integer
  lstVsi.Clear
  With pod.Usluzbenci
    For i = 1 To .Prestej
      With .Element(i)
        lstVsi.AddItem .ID & ": " & .Ime & ", " & .Placa
      End With
    Next
  End With
End Sub
```

Za sprehod uporabimo dva člana razreda Usluzbenci: Prestej vrne skupno število elementov, metoda Element pa vrne element, naslovljen bodisi z zaporedno številko bodisi z opisnim ključem, povsem enako kakor pri vgrajenih zbirkah (oglejmo si kodo metode Element, kot parameter sprejme tip Variant, ki ga posreduje naprej vgrajeni zbirki). Paziti moramo na meje intervala. Pri vgrajenih zbirkah potekajo od enice do vrednosti lastnosti Count (v našem primeru je lastnost Count *ovita* z metodo Prestej).

Sedaj lahko zapišemo tudi podprogram, ki poskrbi za odstranitev izbranega elementa:

```
Private Sub cmdOdstrani_Click()
  If lstVsi.ListIndex > -1 Then
    pod.Usluzbenci.Odstrani lstVsi.ListIndex + 1
    OsveziSeznam
  End If
End Sub
```

Enostavno odstranimo element, katerega zaporedna številka je za eno večja od zaporedne številke vrstice na odprtem seznamu (te se namreč začnejo z ničlo).

Sprehod

Kaj še manjka, da bo naša nova zbirka popolna? Pri osveževanju seznama uporabljamo zanko For Next, večkrat pa smo že omenili, da je pri zbirkah sprehod po vseh elementih mogoč tudi z zanko For Each.

Če želimo, da je naše lastne zbirke mogoče uporabiti tudi znotraj zanke For Each, moramo v razredu priskrbeti posebej označeno metodo NewEnum, ki vrača *naštevni* predmet (*enumerator*) zbirke. Naštevnih predmetov v kodi visual basica ne moremo zapisati, kar pa ni nobena ovira, saj lahko uporabimo pripravljeni števec vgrajenega razreda Collection.

Definicijo razreda Usluzbenci bomo dopolnili z novo javno metodo:

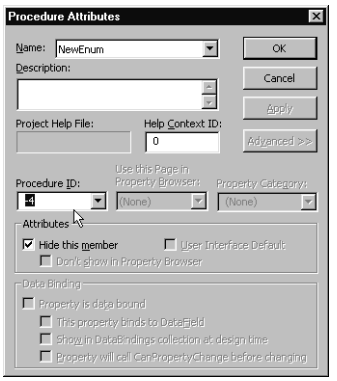
```
Public Function NewEnum() As IUnknown
  Set NewEnum = mcolZbirka.[_NewEnum]
End Function
```

Preprosto bomo vrnili naštevni predmet razreda Collection, ki ga predstavlja naša zasebna predmetna spremenljivka. Ta predmet je dostopen prek skrite metode _NewEnum. Prav začetni podčrtaj je oznaka njene zakritosti. Zaradi takega imena jo moramo objeti z oglatima oklepajema ([]). Podatkovni tip, ki ga vrača funkcijska metoda, mora biti IUnknown. S tem pa naše delo še ni končano. Rekli smo, da mora biti metoda NewEnum posebej označena.

Označevanje metode NewEnum, ki vrača naštevni predmet zbirke, dosežemo z ukazom **Procedure Attributes**... iz glavnega menuja **Tools**. S seznama **Name** izberemo ime metode NewEnum. Kliknemo gumb **Advanced**, da razpremo

pogovorno okno. V skupini Attributes vklopimo stikalo **Hide this member**. S tem bo metoda skrita v razredni knjižnici. Metode, ki vračajo naštevni predmet, pa morajo biti še posebej označene. Na kombinirani seznam **Procedure ID** moramo vnesti število –4 (minus štiri). Tako oznako zahteva zanka For Each. S tem smo delo končali in lahko potrdimo nastavitve, tako da kliknemo OK.

Slika 17-14: Določanje atributov metode NewEnum



Na obrazec lahko sedaj zapišemo tudi bolj berljivo obliko procedure osveževanja seznama:

```
Private Sub OsveziSeznam()
Dim usl As Usluzbenec
  lstVsi.Clear
  For Each usl In pod.Usluzbenci
    With usl
      lstVsi.AddItem .ID & ": " & .Ime & ", " & .Placa
    End With
  Next
End Sub
```

Z nekaj truda smo razvili zelo zmogljiv predmetni model, ki ga lahko poljubno širimo.

> **NAMIG** Programsko kodo predmetnega modela najdete na priloženi plošči CD-ROM, v mapi \Primeri\4-Razredi\1-Lastni razredi\5-Predmetni model\.

Pomočniki

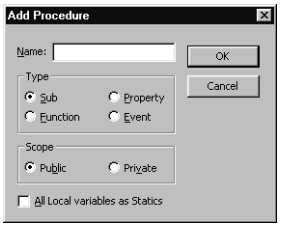
Videli smo, da je kodiranje razvejene predmetne strukture dolgotrajno opravilo, ki vključuje definicije celega kupa razrednih modulov in določanje njihove medsebojne odvisnosti. Pri kodiranju nam lahko pomagata dva pomočnika. Prvi omogoča olajšano kodiranje članov razreda, drugi pa nam oblikuje celoten skelet

poljubno zapletenega predmetnega modela[59]. Preden si ogledamo njuno demonstracijo, shranimo dejavni projekt in odprimo novega vrste »Standard EXE«. V projekt dodajmo nov razredni modul.

Dodajanje člana

Pri kodiranju razrednih modulov nam pomaga ukaz **Add Procedure**..., ki ga najdemo v glavnem menuju **Tools**. Prikaže pogovorno okno, ki olajša pripravo skeleta novega člana razreda. Vidno je na spodnji sliki:

Slika 17-15: Pogovorno okno za dodajanje novega člana razreda



V vnosno polje Name vnesemo ime novega člana in izberemo njegovo vrsto ter doseg (javni ali zasebni). Po želji lahko izberemo, da so vse zasebne spremenljivke v pripadajočih podprogramih statične. Po kliku OK dobimo skelet bodočih podprogramov, ki ga moramo dopolniti s programsko kodo. Orodje bi bilo bolj uporabno, ko bi nam pri kodiranju lastnosti pripravilo tudi zasebno proceduro, ki jo hrani.

Graditelj razredov

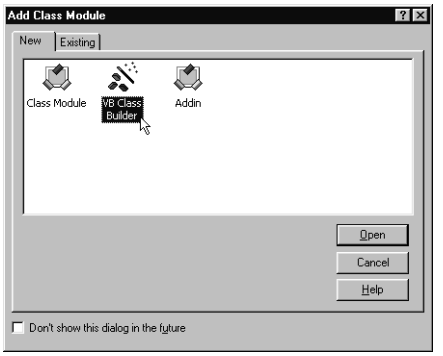
V boljših različicah visual basica (Professional in Enterprise) lahko uporabimo mnogo zmogljivejše orodje, ki omogoča razvoj skeleta celotnega predmetnega modela. Imenuje se Graditelj razredov (*Class Builder*). Ponavadi ga izberemo pri povsem praznem projektu, ker lahko le tako nadzoruje vse dodane razredne module.

Graditelj razredov je na voljo na seznamu predlog, ki se prikaže ob izbiri ukaza **Add Class Module**... iz glavnega menuja **Project**. Njegova ikona je vidna na sliki 17-16.

> **NAMIG** Prikaz seznama predlog lahko onemogočimo z izbiro ustreznega stikala. Če se pri dodajanju novega razrednega modula seznam predlog ne prikaže, preverimo nastavitev **Show Templates For**, ki se skriva pod jezičkom **Environment** pogovornega okna ukaza **Options** iz glavnega menuja **Tools**. V tej skupini se skriva več stikal, ki določajo prikaz seznama predlog za različne elemente projekta.

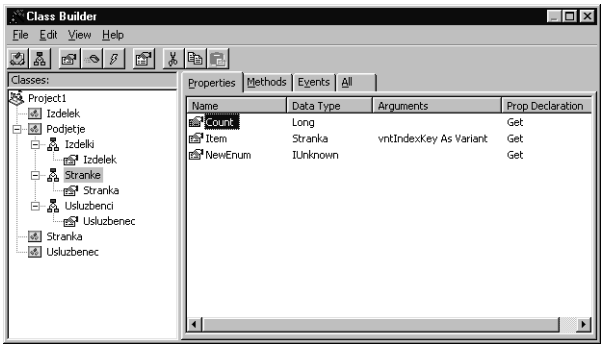
[59] To orodje je na voljo le pri boljših različicah visual basica.

Slika 17-16: Izbor Graditelja razredov



Z Graditeljem razredov lahko pripravimo poljubno razvejen predmetni model. Ko se nam zdi, da je model dobro definiran, izberemo ukaz Update Project, ki ustvari vse razredne module in vanje doda skelet programske kode z nekaj komentarji, ki nam pomagajo pri pisanju programske kode. Potem lahko graditelj zapremo in se posvetimo programski kodi. Nove člane razrednih modulov lahko dodajamo tudi ročno, vendar se moramo izogibati ročnemu dodajanju novih razrednih modulov. Graditelja razredov lahko poženemo znova in samodejno bo prebral vse spremembe v programski kodi in jih prikazal v shemi predmetnega modela.

Slika 17-17: Uporaba graditelja



Poimenovanje

Pri izgradnji predmetnega modela imamo opraviti s celo vrsto razredov in njihovih članov. Podobno kakor pri projektih z več obrazci in gradniki je tudi pri projektih z več razrednimi moduli ustrezno poimenovanje nadvse pomembno. Z njim si olajšamo kodiranje, še posebej pa poznejšo uporabo in vzdrževanje razredov.

Tudi pri poimenovanju razredov ni tako pomembno, kakšna pravila poimenovanja izberemo, pomembno je, da se teh pravil dosledno držimo v vseh projektih, ki jih bomo razvili. Če razrede razvijamo skupaj z drugimi programerji, moramo pravila poimenovanja določiti skupaj in se jih držati vsi. Naštejmo nekaj napotkov za določanje pravil poimenovanja:

- Uporabljajmo daljša opisna imena in ne skrajšav, ki jih že drugi dan ne bomo več razumeli.
- S predlogi določimo zasebne in javne dele razredov. Za določanje zasebnih delov lahko posvojimo predlog »m«, ki ga predlaga tudi Microsoft.
- Če želimo ohraniti skladnost s pripravljenimi predmeti, bomo javne člane razreda poimenovali z veliko začetnico, brez posebnih predlogov, ki bi opisovali njihov podatkovni tip. Morda je bolje, da se tej skladnosti odrečemo in natančneje opišemo tudi tip.
- Podobna opomba velja tudi za parametre metod. Če želimo ohraniti skladnost z obstoječimi metodami, jih bomo poimenovali z veliko začetnico, brez opisa tipa. Tu se skladnosti lahko tudi držimo, saj nam urejevalnik kode s sprotnimi namigi ponazarja, kakšnega podatkovnega tipa je ustrezen parameter.
- Pri parameterih metod, kjer je le mogoče, uporabljajmo poimenovane oziroma opcijske parametre s predpisano privzeto vrednostjo.
- Za lažjo rabo razreda najpogosteje rabljenega člana označimo kot privzetega.

In najpomembnejše: četudi nam opisna imena vzamejo več časa pri vnašanju kode, se spomnimo, da vnos kode obsega le 10 % razvojnega časa, 90 % pa ga porabimo za odkrivanje lastnih napak.

18. POGLAVJE

Gradniki

Z razvojem lastnih razrednih modulov smo se dotaknili morda najzmogljivejše zmožnosti visual basica – priprave lastnih *gradnikov* ActiveX. Gradniki (*components*) omogočajo vnovično izrabo pripravljene programske kode, oblikovane v razrede. Ker svoje storitve ponujajo zunanjemu svetu, jim pogosto rečemo tudi strežniki (*servers*), programi, ki izkoriščajo njihove storitve, pa so odjemalci (*clients*).

Zunanji gradniki so bili na voljo že za starejše različice visual basica, le da so se takrat imenovali »knjižnice VBX«. Postali so silno priljubljeni in kmalu ni bil samo visual basic tisti, ki jih je znal uporabljati. Knjižnice VBX je posvojila večina orodij za razvoj okenskih programov. Z razvojem operacijskega sistema Okna in posebnega protokola za povezovanje predmetov OLE (*Object Linking and Embedding*) so gradniki postali »knjižnice OCX« in kmalu dobili novo, tržnim namenom prilagojeno ime – ActiveX.

V visual basicu lahko oblikujemo več vrst gradnikov. Gradnike, ki odjemalcem ponujajo v glavnem storitve svoje kode, razvijemo v obliki strežnikov ActiveX. Ti nimajo posebnega vizualnega dela, čeprav lahko vanje vključimo poljubno število obrazcev. Posebna oblika strežnikov ActiveX so samostojni programi, ki so hkrati tudi vir izpostavljenih razredov za avtomatizacijo. Danes je takih že večina bolj priljubljenih namenskih programov za operacijske sisteme Okna, denimo vsi večji programi v zbirki Microsoft Office. Na prvi pogled se vedejo kot navadni programi, če pa imamo na voljo prava razvojna orodja, lahko izkoriščamo tudi njihove izpostavljene razrede.

Druga vrsta gradnikov so zunanji gradniki uporabniškega vmesnika. Tem že od začetka pripada grafični vmesnik, namenjeni pa so predvsem uporabi v orodjih za hitro razvijanje programov (RAD, *Rapid Application Development*). Tovrstne gradnike lahko z izpostavljenimi lastnostmi vizualno sestavljamo v zmogljivejše vmesnike, javne metode pa omogočajo njihovo učinkovito rabo s čim manj zapisane programske kode.

Strežniki

S strežniki ActiveX lahko poljubno funkcionalnost, ki jo razvijemo z visual basicom, izpostavimo in ponudimo vsem drugim programom v sistemu. Strežnike lahko

uporabljamo v novih projektih v visual basicu ali pa v kateremkoli drugem programu, ki podpira programirano avtomatizacijo (npr. Microsoft Word, Corel Draw, Borland Delphi itn.). Posebna oblika strežnika je lahko tudi samostojen program, ki izpostavlja svoje razrede zunanjemu svetu.

Posamezni izvod programa, ki se izvaja v operacijskem sistemu Okna, imenujemo *proces*. Ta ima svoj naslovni prostor, v katerem je shranjena njegova koda ter vrednosti spremenljivk, ki jih uporablja pri svojem delu. Prednost posebnega naslovnega prostora za vsak proces je omogočena *predkupna večopravilnost* (*preemptive multitasking*) in večja trdoživost sistema. Več procesov se lahko izvaja hkrati, če pa posamezen proces razpade, to na druge procese ne vpliva neposredno.

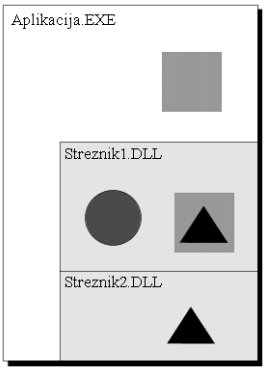
Strežniki ActiveX se lahko izvajajo znotraj ali zunaj naslovnega prostora procesa, ki je njihov odjemalec. Znotrajprocesni strežniki so manjši in hitrejši, zunajprocesni pa omogočajo asinhrone klice, ponujajo večjo trdoživost in omogočajo izdelovanje samostojnih programov.

Ker je namen strežnikov predvsem ponujanje svojih storitev drugim programom, je dobro načrtovanje osnova uporabnega gradnika. Izdelavo novega strežnika bi lahko razčlenili na naslednje korake:

- Ugotavljanje zmogljivosti, ki jih želimo ponuditi prek strežnika. Izbira znotraj- ali zunajprocesne oblike strežnika.
- Razčlenitev zmogljivosti v predmetni model (predmetna analiza). Določitev javnega vmesnika, dogodkov in zasebne komunikacije med razredi. Poleg razrednih modulov bomo morda potrebovali tudi samostojne module s kodo.
- Načrtovanje morebitnih obrazcev za vizualno komunikacijo z uporabnikom.
- Kodiranje razredov ter morebitnih obrazcev in samostojnih kodnih modulov.
- Preizkušanje in dopolnjevanje strežnika s testnim projektom znotraj razvojnega okolja visual basica (znotrajprocesni) ali novega izvoda visual basica (zunajprocesni strežnik).
- Prevajanje v knjižnico DLL oziroma v izvršljivo datoteko EXE in dokončno preizkušanje zunajrazvojnega okolja visual basica.
- Morebitna priprava namestitvene različice.

Znotrajprocesni strežnik

Znotrajprocesni strežnik (*in-process server*), imenovan tudi »ActiveX DLL«, se izvaja znotraj naslovnega prostora procesa. Njegov odjemalec je lahko sam proces ali pa drug znotrajprocesni strežnik, ki je prav tako na istem naslovnem prostoru.



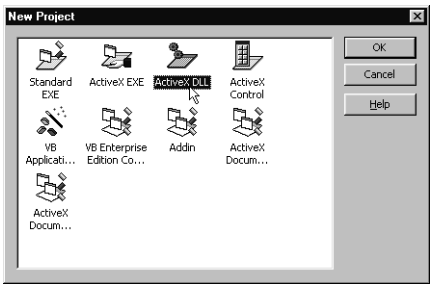
Slika 18-1: Znotrajprocesni strežnik se izvaja v naslovnem prostoru procesa in daje svoje predmete na voljo drugim znotrajprocesnim strežnikom ter sami aplikaciji.

Znotrajprocesni strežniki se odzivajo hitreje in so kompaktnejši, vendar so močno povezani z odjemalcem. Zato uporaba njihovih storitev začasno zaustavi izvajanje preostale kode, kar posredno vpliva tudi na trdoživost izdelka.

Za zgled bomo razvili strežnik z definicijo pogovornega okna. To pogovorno okno bomo lahko prikazali in uporabili v vsakem drugem projektu. Tak strežnik je mogoče razširiti s poljubnim številom prilagodljivih pogovornih oken. Tako si lahko zagotovimo osnovo uporabniškega vmesnika za vse naše projekte. Pa se lotimo dela.

Vzpostavitev projekta

Kodiranje novega znotrajprocesnega strežnika začnemo z izbiro novega projekta vrste »ActiveX DLL«. Tak projekt na začetku sestavlja le en razredni modul, ki mu lahko nastavimo ime in posebno lastnost **Instancing**, ki določa vzpostavljanje posameznega razreda strežnika.



Slika 18-2: Izbira novega projekta vrste »ActiveX DLL«

Poimenujmo nov razredni modul »Dialog«. Pri lastnosti Instancing nam privzeta nastavitev ne odgovarja najbolj. Oglejmo si opis razpoložljivih možnosti vzpostavljanja posameznega razreda znotrajprocesnega strežnika:

Vzpostavljanje znotrajprocesnega gradnika	Opis
1 – Private.	Druge aplikacije nimajo dostopa do tega razreda, uporabljamo ga lahko le znotraj drugih modulov v visual basicu.
2 – PublicNotCreatable.	Druge aplikacije ta razred lahko uporabljajo, ne morejo pa ustvarjati njegovih novih izvodov. Za vzpostavitev izvoda moramo poskrbeti v strežniku.
5 – MultiUse.	Druge aplikacije lahko ustvarijo neomejeno število izvodov tega razreda. To je tudi privzeta nastavitev.
6 – GlobalMultiUse.	Podobno prejšnji nastavitvi, le da je doseg izpostavljenega razreda globalen. Aplikacije lahko uporabljajo njegove zmogljivosti brez vzpostavitve posebnega izvoda razreda, saj se ta samodejno ustvari ob prvi uporabi lastnosti ali metode (podobno implicitni stvaritvi). Klic metode je npr. enak klicu globalne funkcije.

V strežniku lahko določimo poljubno število razredov. Z lastnostjo Instancing lahko nekatere razrede skrijemo, druge pa izpostavimo na različne načine. Za prikaz samodejnega vzpostavljanja, ki v marsičem olajša uporabo gradnika, bomo našemu edinemu razredu v strežniku nastavili lastnost Instancing na vrednost »6 – GlobalMultiUse«.

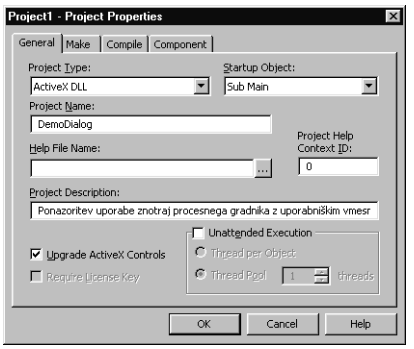
Slika 18-3: Določanje lastnosti razrednega modula

Naslednji korak je določitev lastnosti samega projekta. V glavnem menuju **Project** izberemo ukaz **Project1 Properties**... Prikaže se pogovorno okno nastavitev projekta, v katerem izberemo jeziček **General** (slika 18-4).

Na seznamu **Project Type** je že izbrana pravilna vrsta projekta, torej »ActiveX DLL«. Seznam **Startup Object** po privzeti nastavitvi kaže na »(none)«, kar določa, da naš strežnik nima posebnega vzpostavitvenega mehanizma. Če želimo ob zagonu strežnika kaj postoriti, s tega seznama izberemo »Sub Main«. V tem primeru bomo morali pozneje projektu dodati samostojen modul s kodo in v njem določiti

zagonsko proceduro. Zaradi spremljanja življenjskega kroga strežnika izberimo to možnost. Vnosno polje **Project Name** mora vsebovati simbolično ime strežnika, ki mora biti enolično, pravilno ime (presledki npr. so izključeni). To ime dejansko določa ime razredne knjižnice, ki omogoča izkoriščanje izpostavljenih razredov v drugih aplikacijah. V našem zgledu bomo ime nastavili na »DemoDialog«. Za konec v vnosno polje **Project Description** vnesemo še daljše opisno ime strežnika. To ime je pomembno, saj se po končanem razvoju in objavi novega gradnika v sistemskem registru prikaže na seznamu razpoložljivih predmetnih knjižnic. Opisane nastavitve so razvidne tudi na naslednji sliki:

Slika 18-4: Določanje lastnosti projekta



Modul s kodo

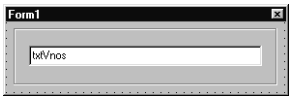
Zaradi izbire zagonske procedure moramo sedaj v projekt vključiti samostojni modul s kodo. Dodamo ga z izbiro ukaza **Add Module** iz menuja **Project**. Modul bo vseboval vzpostavitveno proceduro in globalno dosegljivo uslužnostno funkcijo, ki nam bo pomagala pri morebitnem razhroščevanju:

```
' Koda samostojnega modula

' Globalna spremenljivka
Private gdteZagon As Date

' Procedura, v kateri se začne izvajanje
Public Sub Main()
  gdteZagon = Now
  Debug.Print "Strežnik " & App.EXEName & " zagnan!"
End Sub

' Funkcija vrne ID, ki rabi za
' enolično označevanje elementov
Public Function NovID() As Long
Static lngID As Long
  lngID = lngID + 1
  NovID = lngID
End Function
```

V zagonski proceduri bomo shranili trenutek vzpostavitve strežnika in izpisali obvestilo v takojšnje okno. Določili smo še globalno dosegljivo funkcijo, ki vrača vedno drugačno označevalno število, s katerim bomo lahko označili in spremljali posamezne elemente. Tovrstno označevanje nam lahko precej pomaga pri iskanju pozabljenih referenc predmetov.

Uporabniški vmesnik

Odločili smo se, da bo naš strežnik imel uporabniški vmesnik. Tega moramo določiti v posebnem modulu, ki pri projektu vrste »ActiveX DLL« ni takoj na voljo. Z ukazom **Add Form**... iz menuja **Project** projektu dodamo nov, prazen obrazec. Določimo mu uporabniški vmesnik po vzoru spodnje slike:



Slika 18-5: Uporabniški vmesnik strežnika

Na obrazec postavimo naslednje gradnike:

Gradnik	Lastnost	Nastavitev
Obrazec (*Form*)	(Name)	FrmDialog
	BorderStyle	4 – Fixed ToolWindow
Okvir (*Frame*)	(Name)	Frame1
Vnosno polje (*TextBox*)	(Name)	txtVnos

Zapisati moramo še nekaj kode, ki bo skrbela za njegovo pravilno vedenje. Najprej bomo poskrbeli za proženje dogodka ob vsaki spremembi v vnosnem polju:

```
' Koda obrazca FrmDialog

Public Event Vnos(strVnos As String)

Private Sub txtVnos_Change()
  RaiseEvent Vnos(txtVnos.Text)
End Sub
```

Ker ne želimo, da bi se obrazec odstranil nenadzorovano, bomo vsak poizkus odstranitve preslikali v skrivanje:

```
Private Sub Form_QueryUnload(Cancel As Integer, UnloadMode As Integer)
  If UnloadMode = vbFormControlMenu Then
    Me.Hide
    Cancel = True
  End If
End Sub
```

Dodali bomo še dve uslužnostni odzivni proceduri. Prva bo poskrbela, da bo pred vnosom označena morebitna že prej prisotna vsebina (tako se vede večina vnosnih polj v pogovornih oknih). Druga procedura bo ustrezno obravnavala tipki Enter in Escape:

```
Private Sub txtVnos_GotFocus()
  With txtVnos
    .SelStart = 0
    .SelLength = Len(.Text)
  End With
End Sub

Private Sub txtVnos_KeyPress(KeyAscii As Integer)
  Select Case KeyAscii
    Case 13
      KeyAscii = 0
      Me.Hide
    Case 27
      KeyAscii = 0
      txtVnos.Text = ""
      Me.Hide
  End Select
End Sub
```

Obrazec je vedno *zasebni* razred projekta. Njegov prikaz in skrivanje bomo v celoti nadzorovali iz razreda Dialog. Zasebni razredi odjemalcem niso dostopni. Izvodov zasebnih razredov tudi nikoli ne smemo posredovati strežniku prek kakega drugega javno izpostavljenega člana (npr. lastnosti).

Razredni modul

Razredni modul Dialog bo ponujal sposobnosti zasebnega obrazca prek svoje metode in dogodka. Najprej zapišimo del splošnih napovedi modula. Zaradi spremljanja posameznih izvodov bomo razredu dodelili zasebno spremenljivko mlngID. Potrebujemo tudi zasebnega člana, ki bo predstavljal prej dokončani obrazec. Ker vmesnika zasebnega razreda ne smemo izpostavljati navzven, bomo dogodek obrazca preslikali v dogodek razreda Dialog.

```
' Razredni modul Dialog

Private mlngID As Long
Private WithEvents mfrmDialog As FrmDialog

Public Event Vnos(strVnos As String)
```

Koda vzpostavitve razreda ne bo nič posebnega. Poskrbeli bomo za začetne vrednosti zasebnih spremenljivk, kar pomeni tudi vzpostavitev zasebnega predmeta razreda FrmDialog, torej obrazca:

```
Private Sub Class_Initialize()
  mlngID = NovID
  Set mfrmDialog = New FrmDialog
  Debug.Print "Ustvarjen predmet, ID " & mlngID
End Sub
```

Odziv na dogodek Terminate bo zglednejši, prikazoval bo pravilno odstranitev obrazca iz pomnilnika. Najprej bomo morali odstraniti njegov vizualni del, nato pa se moramo znebiti še njegove reference s postavitvijo predmetne spremenljivke na neveljavno vrednost (pregled življenjskega kroga obrazca smo obravnavali v prejšnjem poglavju).

```
Private Sub Class_Terminate()
  Unload mfrmDialog
  Set mfrmDialog = Nothing
  Debug.Print "Uničen predmet, ID " & mlngID
End Sub
```

In za konec še izpostavljena funkcionalnost razreda. Razred bo uporaben skozi javno metodo Dialog, ki bo sprejela dva neobvezna parametra. Prvi bo določal privzeto besedilo v vnosnem polju, drugi pa bo logična spremenljivka, ki bo določala modalnost okna. Glede na posredovane parametre bomo poskrbeli za vzpostavitev vnosnega polja in ustrezen prikaz okna:

```
Public Function Dialog(Optional ByVal strBesedilo As String = "", _
    Optional ByVal blnModalno As Boolean = True) As String
  With mfrmDialog
    .txtVnos.Text = strBesedilo
    If blnModalno Then
      .Caption = "Modalno"
      .Show vbModal
      Dialog = .txtVnos.Text
    Else
      .Caption = "Nemodalno"
      .Show
    End If
  End With
End Function
```

Druga procedura bo zasebna, vendar proži navzven viden javni dogodek, ki je v bistvu preslikava istoimenskega dogodka samega obrazca. Kot smo že omenili, elementov uporabniškega vmesnika ne smemo *neposredno* izpostavljati skozi javne člane preostalih razredov:

```
Private Sub mfrmDialog_Vnos(strVnos As String)
  RaiseEvent Vnos(strVnos)
End Sub
```

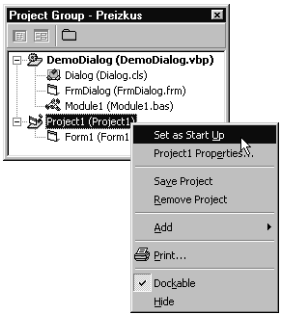
Tako smo dokončali svoj prvi znotrajprocesni strežnik, gradnik ActiveX. Preostane nam, da ga preizkusimo v razvojnem okolju ter prevedemo in vzpostavimo v sistemu. Pred tem še varno shranimo vse datoteke.

Preizkus

Znotrajprocesne gradnike ActiveX lahko preizkusimo, ne da bi zaprli dejavni projekt. Visual basic omogoča hkratno delo z več odprtimi projekti. Testni projekt bomo enostavno dodali v skupino. V glavnem menuju **File** izberemo ukaz **Add**

Project... (pozor, *Add* in ne New ali Open). Odpre se seznam predlog, s katerega izberemo običajni projekt vrste »Standard EXE«. Projektno okno je sedaj videti nekako takole:

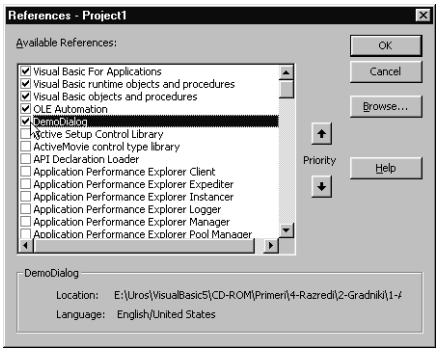
Slika 18-6: Projektno okno preizkusnika in preizkušanca



V projektnem oknu izberemo svež projekt in ga z ukazom **Set as Start Up** iz njegovega priročnega menuja določimo kot zagonski projekt (kakor prikazuje zgornja slika). Ime zagonskega projekta je v projektnem oknu izpisano krepko.

Če bi sedaj odprli razredni brskalnik in v njem poizkušali najti razred Dialog, nam to ne bi uspelo. Najprej moramo poskrbeti za vklop novo razvite razredne knjižnice. Kot vemo, razredne knjižnice vklapljamo z ukazom **References**... iz glavnega menuja **Project**. Čeprav preizkušana razredna knjižnica še ni ustrezno napovedana v sistemskem registru, razvojno okolje poskrbi, da je že na seznamu pogovornega okna. Visual basic je celo tako ustrežljiv, da njeno ime uvrsti takoj za vsemi že vključenimi knjižnicami, kot je razvidno s spodnje slike:

Slika 18-7: Vklop razredne knjižnice



Ko preizkušano razredno knjižnico vklopimo, so nam v programski metodi dostopni njeni razredi. No, pravzaprav samo en razred: Dialog.

Pred zapisom kode si pripravimo uporabniški vmesnik obrazca. Nanj namestimo dva ukazna gumba in jima pustimo privzeti imeni. Sedaj se lotimo kodiranja. Najprej bomo napovedali zasebno predmetno spremenljivko, ki bo kazala na

predmet razreda DemoDialog.Dialog, sposoben proženja dogodkov. Ime razreda bo zapisano v polni obliki, ki določa tako razredno knjižnico kakor ime razreda. Kolikor nam je znano, v drugih razrednih knjižnicah razreda Dialog ni, če pa bi slučajno bil, bi s tako zapisanim imenom razrešili imenski konflikt[60]:

```
' Preizkusni obrazec

Private WithEvents dlg As DemoDialog.Dialog
```

> **NAMIG** Za razreševanje morebitnih imenskih konfliktov je poskrbljeno že z vrstnim redom predmetnih knjižnic, kot je določen v pogovornem oknu ukaza References. Knjižnice se vedno pregledujejo od vrha navzdol, dokler se ne najde prvi razred, s čimer je iskanja konec. Zato to pogovorno okno ponuja tudi gumbe za preurejanje prioritete predmetnih knjižnic.

Sedaj se lotimo uporabe prvega gumba. Ko bomo kliknili gumb, bomo preverili, ali je bil predmet že ustvarjen, in ga bomo po potrebi ustvarili. Zatem bomo poklicali njegovo javno metodo Dialog, ki bo prikazala nemodalno pogovorno okno z vsebino, enako napisu na gumbu:

```
Private Sub Command1_Click()
  If dlg Is Nothing Then
    Set dlg = New DemoDialog.Dialog
  End If
  dlg.Dialog Command1.Caption, False
End Sub
```

Ker smo okno prikazali nemodalno, se bo izvajanje kode nadaljevalo tudi po njegovem prikazu. Za uporabo dialoga bo skrbel prestrezovalnik dogodka Vnos. Ob vsakem dogodku bo ustrezno popravil napis na gumbu:

```
Private Sub dlg_Vnos(strVnos As String)
  Command1.Caption = strVnos
End Sub
```

In drugi gumb? Drugi gumb bo prikazoval vpliv posebnega načina vzpostavitve razreda Dialog, ki ga določa njegova lastnost Instancing. Ker smo jo nastavili na vrednost »6 – GlobalMultiUse«, so vsi njegovi javni člani dostopni tudi neposredno, brez sklicevanja na obstoječi izvod razreda. Ob klicu metode Dialog se bo ustrezen razred izdelal samodejno:

```
Private Sub Command2_Click()
  Command2.Caption = Dialog(Command2.Caption)
End Sub
```

Se bo za obravnavo odzivnih procedur obeh gumbov uporabil isti izvod razreda Dialog? Odgovor na to vprašanje bomo lahko prebrali v takojšnjem oknu, ko bomo

[60] Imena razplozljivih razredov lahko preverimo s predmetnim brskalnikom.

zgled preizkusili. Verjetno bo najbolje, če se po razviti kodi sprehodimo korak za korakom z ukazom **Debug ➪ Step Into**.

> **NAMIG** Če se po zagonu ne prikaže obrazec preizkusnega projekta, verjetno niste nastavili pravega zagonskega projekta. Preverite vsebino projektnega okna.

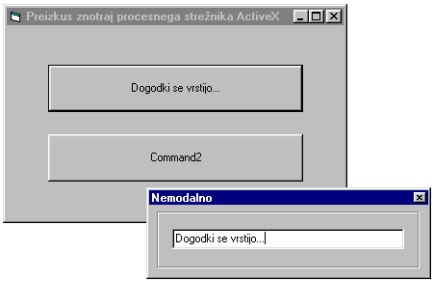
Ob zagonu se prikaže obrazec preizkusnega projekta, takojšnje okno pa je prazno. Znotrajprocesni strežniki ActiveX se uporabljajo dinamično. Strežnik se bo zagnal šele ob prvi uporabi njegove storitve.

Ko kliknemo prvi gumb, se prikaže pogovorno okno strežnika, v takojšnjem oknu pa lahko preberemo:

```
Strežnik DemoDialog zagnan!
Ustvarjen predmet, ID 1
```

Takoj lahko preizkusimo odziv na verižno povezane dogodke Change ➪ Vnos ➪ Vnos. Ob kakršnemkoli vnosu v polje pogovornega okna se spremeni tudi napis na gumbu.

Slika 18-8: Odziv na spremembe v vnosnem polju



Ker je pogovorno okno prikazano nemodalno, lahko kadarkoli kliknemo tudi drugi gumb. Sedaj se prikaže novo, modalno prikazano pogovorno okno, v takojšnjem oknu pa nas čaka odgovor na zgoraj zastavljeno vprašanje:

```
Ustvarjen predmet, ID 2
```

Da, verjetno bi lahko vedeli že od prej. Klic metode pomeni implicitno stvaritev novega, *globalnega* izvoda razreda Dialog. Ta izvod se uporabi pri vseh nadaljnjih neposrednih klicih metod razreda, vendar ga ne moremo uporabiti za prestrezanje dogodkov. Če želimo uporabiti dogodke, moramo določiti predmetno spremenljivko, tako kakor pri prvem gumbu. Pri koračnem izvajanju bomo ugotovili, da tudi globalni izvod proži dogodke, težava je v tem, da v kodi nimamo njegove reference.

Na to dejstvo velja še posebej opozoriti. Ker v kodi ne poznamo reference, ki vzdržuje drugi, globalni izvod razreda pri življenju, se ga tudi ne moremo

neposredno znebiti. V našem projektu s tem ni težav, saj se novi izvodi ne bodo več ustvarjali. Povsem drugače bi bilo, ko bi se tovrstni »izgubljeni« izvodi pojavljali kar naprej. Sčasoma bi lahko precej izčrpali razpoložljiva sistemska sredstva in učinkovitost programa bi se bistveno zmanjšala.

> **NAMIG** Čeprav se globalni izvod v preizkusnem projektu ustvari le enkrat, to ne pomeni, da ga lahko uporabimo samo v enem odjemalcu. Ravno nasprotno, posebej izdelan globalen izvod bo pripadel vsakemu odjemalcu, ki bo uporabljal naš strežnik.

Preverimo še, kaj se zgodi, če zapremo prikazani pogovorni okni in ju znova prikličemo. Nič novega, uporabljata se oba prej ustvarjena izvoda. Preizkus končamo z zapiranjem obrazca preizkusnega projekta. V takojšnjem oknu lahko preberemo, da sta skupaj z njim umrla tudi oba izvoda predmeta Dialog:

```
Uničen predmet, ID 2
Uničen predmet, ID 1
```

Prevajanje in napoved

Ko je novo razviti strežnik temeljito preizkušen v razvojnem okolju, lahko poskrbimo za njegovo prevajanje v deljeno knjižnico DLL. To opravilo ni prav nič drugačno od prevajanja v samostojni program, le da je končni izid deljena knjižnica DLL. Med prevajanjem visual basic tudi poskrbi, da se novo ustvarjeni strežnik pravilno napove v sistemskem registru in s tem postane dostopen vsem morebitnim odjemalcem v operacijskem sistemu.

V našem primeru moramo najprej izbrati ustrezen projekt v projektnem oknu, da ne bi po pomoti prevajali preizkusnega projekta. V projektnem oknu torej izberemo projekt DemoDialog in zatem ukaz **Make DemoDialog.dll** iz glavnega menuja **File**. Po koncu prevajanja lahko projekt DemoDialog odstranimo iz skupine. V njegovem priročnem menuju znotraj projektnega okna izberemo ukaz Remove Project. Visual basic nas bo opozoril, da se na ta projekt sklicujemo v drugem projektu, vendar lahko to opozorilo prezremo. Ob vnovičnem zagonu testnega projekta bo sam prebrskal register in poizkušal najti ustrezno knjižnico DLL.

Če želimo, se lahko vrnemo v nastavitev uporabljenih predmetnih knjižnic in izklopimo projekt v visual basicu. Nekje na seznamu mora zdaj biti tudi referenca na prevedeno knjižnico DLL. Njeno ime je odvisno od daljšega opisnega imena, ki smo ga vnesli pri vzpostavitvi projekta strežnika.

> **NAMIG** Pri nameščanju prevedenega gradnika v drug sistem je za napoved v sistemskem registru treba poskrbeti naknadno. Pripomoček Application Setup Wizard zna poskrbeti za vse potrebno.

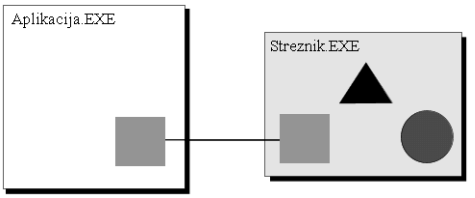
Prikaz nemodalnih pogovornih oken skozi znotrajprocesne strežnike zahteva komunikacijo z osnovno sporočilno zanko odjemalca (*client message loop*). Vsi odjemalci tega ne podpirajo. Strežnik lahko analizira odjemalca prek posebne lastnosti predmeta App, imenovane **NonModalAllowed**. Nemodalna okna je mogoče prikazovati le, če ima ta lastnost resnično vrednost.

Programsko kodo razvitega strežnika in preizkusnega projekta najdete na priloženi plošči CD-ROM v mapi \Primeri\4-Razredi\2-Gradniki\1-ActiveX DLL.

Zunajprocesni strežnik

Zunaj procesni strežnik (*out-of-process server*), imenovan tudi »ActiveX EXE«, se izvaja v svojem naslovnem prostoru. Je proces sam zase in lahko predstavlja tudi samostojno, zaokroženo aplikacijo. Zato je ves pogovor med odjemalcem in strežnikom *asinhron*. Odjemalec lahko strežniku naloži določeno opravilo in nadaljuje svoje delo. Strežnik ga po koncu dela obvesti o končani nalogi. Koda je tako lahko precej učinkovitejša, proces komunikacije pa je počasnejši kakor pri rabi znotrajprocesnih strežnikov.

Slika 18-9: Zunajprocesni gradnik se izvaja v svojem naslovnem prostoru, zato omogoča asinhrono komuniciranje.

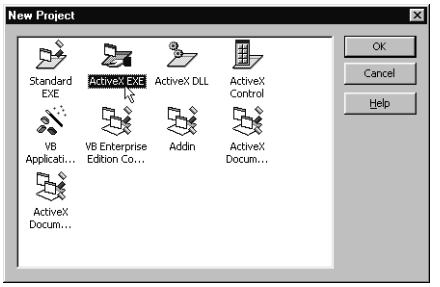


Tudi pri razvoju zunajprocesnih gradnikov veljajo enaka pravila, kakor smo jih opisali pri spoznavanju s strežniki ActiveX. Seveda lahko pri ugotavljanju zmogljivosti, ki jih želimo ponuditi prek zunajprocesnega gradnika, upoštevamo njegove značilnosti, kot sta samostojnost in asinhronost.

Najprej si bomo ogledali posebnosti, ki jih srečamo pri prikazu obrazcev zunajprocesnega strežnika.

Vzpostavitev projekta

V galeriji različnih vrst projektov, ki se prikaže po izbiri ukaza **New Project**, iz glavnega menuja **File** izberemo nov projekt vrste »ActiveX EXE«.



Slika 18-10: Izbira novega projekta vrste »ActiveX EXE«

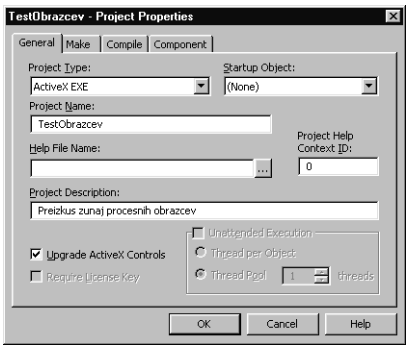
Tudi ta projekt je na začetku sestavljen iz enega samega razrednega modula, ki mu določimo ime in način vzpostavljanja. Večino možnih nastavitev vzpostavljanja razreda smo spoznali že, ko smo obravnavali znotrajprocesne strežnike, tu pa si oglejmo še dve možnosti, ki sta na voljo le pri zunajprocesnih strežnikih. To sta:

Posebni vrednosti lastnosti Instancing za zunajprocesne strežnike	Opis
3 – SingleUse	Druge aplikacije lahko ustvarjajo predmete iz tega razreda, vendar vsak izvod razreda požene nov izvod zunajprocesnega strežnika.
4 – GlobalSingleUse	Podobno zgornji nastavitvi, le da je vmesnik razreda mogoče doseči brez predhodne eksplicitne stvaritve njegovega izvoda.

V našem zgledu bomo lastnost Instancing ohranili na privzeti nastavitvi (5 – MultiUse), spremenili pa bomo ime razreda (lastnost Name) iz privzete oblike »Class1« v zgovornejšo obliko »Obrazec«.

Sedaj nastavimo lastnosti samega projekta. V glavnem menuju **Project** izberemo ukaz **Project1 Properties**... Prikaže se pogovorno okno nastavitev projekta, v katerem izberemo jeziček **General**.

Na seznamu **Project Type** je že izbrana pravilna vrsta projekta, torej »ActiveX EXE«. Seznam **Startup Object** po privzeti nastavitvi kaže na »(none)«, kar določa, da naš strežnik nima posebnega vzpostavitvenega mehanizma. Za naš preizkus je to povsem ustrezno. Pomembna je vsebina vnosnega polja **Project Name**, ki določa simbolično ime razredne knjižnice. V našem zgledu bomo ime nastavili na »TestObrazcev«. Na koncu izpolnimo še opisno ime strežnika. V vnosno polje **Project Description** vnesimo »Preizkus zunajprocesnih obrazcev«.



Slika 18-11: Določanje lastnosti projekta

Preizkusni strežnik

Naš prvi zunajprocesni strežnik bo ponazarjal značilnosti prikaza oken, zato mu dodajmo obrazec. V menuju **Project** izberimo ukaz **Add Form**... in iz galerije izberemo običajni obrazec. Dodajmo mu še ukazni gumb ter določimo naslednje lastnosti:

Gradnik	Lastnost	Nastavitev
Obrazec (*Form*)	(Name)	FrmTest
Ukazni gumb (*CommandButton*)	Caption	Zapri

Gumbu priredimo še naslednjo odzivno proceduro za dogodek Click:

```
Private Sub Command1_Click()
  Unload Me
End Sub
```

Kot že vemo, so vsi obrazci vedno zasebni razredi strežnika. Njihove zmožnosti moramo izpostaviti *posredno*, prek javnega razreda. Preselimo se v okno s kodo razrednega modula »Obrazec«. Najprej mu dodajmo javni naštevni tip:

```
Public Enum obrModalnost
  obrModalen = vbModal
  obrNemodalen = vbModeless
End Enum
```

Vsi javni naštevni tipi, določeni v razrednih modulih, se dodajo v globalni del razredne knjižnice in so neposredno dostopni v vseh aplikacijah, ki to razredno knjižnico vključujejo. Morda se zdi nepotrebno z lastnim naštevnim tipom ovijati obstoječe vgrajene konstante. No, gradniki, razviti v visual basicu, so lahko uporabljeni v najrazličnejših namenskih programih, ki podpirajo avtomatizacijo. V teh programih pa konstanti vbModal in vbModeless morda nista določeni. Uporaba naštevnih tipov, ki jih določimo v samem razredu, je najbolj prilagodljiva rešitev.

Razred rabi le za prikaz posebnih lastnosti zunajprocesnega strežnika, zato bo vseboval samo eno javno metodo. Njena koda se glasi:

```
Public Sub Prikaz(Optional Modalnost As obrModalnost = obrModalen)
Dim frm As New FrmTest
  If Modalnost = obrModalen Then
    frm.Caption = "MODALEN!"
  Else
    frm.Caption = "NEMODALEN!"
  End If
  frm.Show Modalnost
End Sub
```

Metoda lahko sprejme parametre, ki določajo modalnost prikaza obrazca. Ker je naš naštevni tip popolnoma združljiv z vgrajenima konstantama, lahko metodi Show posredujemo kar vrednost tega parametra, ki pa mora imeti določeno privzeto vrednost.

S tem smo razvoj strežnika končali. Najprej ga poizkusno poženemo z ukazom **Start With Full Compile** iz glavnega menija **Run**. Če se prevede in požene brez težav, končamo njegovo izvajanje in pripravimo še prevedeno različico v datoteki EXE, pri čemer nam pomaga ukaz **Make TestObrazcev.EXE** iz glavnega menuja **File**. Postopek prevajanja poskrbi tudi za registracijo novega strežnika v sistemskem registru.

> **NAMIG** Strežnike je najbolje preizkušati s popolnim prevajanjem. Vedno ga lahko dosežemo z ukazom Start With Full Compile. S posebno nastavitvijo dosežemo, da se tudi običajni zagon z ukazom Run vede enako kakor prej omenjeni ukaz. V splošnih nastavitvah prek ukaza **Tools ➪ Options**... izberemo jeziček **General** in v skupini **Compile** izklopimo stikalo **Compile On Demand**. Zagon programov bo sedaj nekoliko daljši, vendar pa nas med izvajanjem ne morejo več presenetiti izvajalne napake, katerih vzrok tiči v nepravilni skladnji kode.

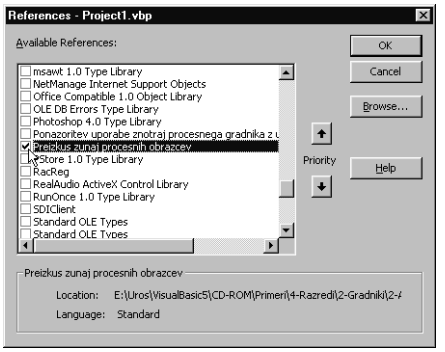
Preizkus

Zunajprocesne strežnike preizkušamo v novem izvodu razvojnega okolja. Prepričajmo se, da je projekt »TestObrazcev« pognan, in zatem zaženimo *nov* izvod visual basica. V njem začnemo nov projekt vrste »Standard EXE«. Na obstoječi obrazec novega projekta dodamo dva ukazna gumba in jima nastavimo lastnosti glede na naslednjo tabelo:

Gradnik	Lastnost	Nastavitev
Ukazni gumb (*CommandButton*)	(Name)	Command1
	Caption	Modalen
Ukazni gumb (*CommandButton*)	(Name)	Command2
	Caption	Nemodalen

Preden so lotimo pisanja programske kode, moramo vključiti razredno knjižnico prej razvitega zunajprocesnega strežnika. Z ukazom **References**... iz glavnega menuja **Project** prikažemo seznam razpoložljivih knjižnic. V njem vključimo knjižnico z opisnim imenom, kot smo ga določili v nastavitvah prejšnjega projekta (»Preizkus zunajprocesnih obrazcev«).

Slika 18-12: Vklop razredne knjižnice



Sedaj je v kodi na voljo razred Obrazec, ki ga uporabimo pri napovedi zasebne predmetne spremenljivke:

```
Private test As Obrazec
```

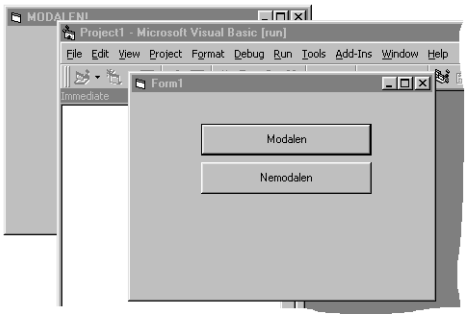
Dodajmo še ustrezne odzivne procedure:

```
Private Sub Form_Load()
  Set test = New Obrazec
End Sub

Private Sub Command1_Click()
  test.Prikaz obrModalen
End Sub

Private Sub Command2_Click()
  test.Prikaz obrNemodalen
End Sub
```

Ob vzpostavitvi obrazca smo vzpostavili tudi predmetno spremenljivko, klik po posameznem gumbu pa se odrazi v dveh različnih načinih prikaza obrazca, ki ga priskrbi zunajprocesni strežnik. Zdaj lahko poženemo tudi drugi projekt in preizkusimo vedenje modalno in nemodalno prikazanih obrazcev.

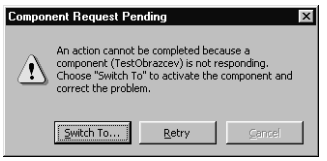
Za začetek kliknimo gumb z napisom »Modalen«, ki prikaže modalno pogovorno okno strežnika. Odvisno od položaja in velikosti oken, vrstnega reda zagona programov in sistemskih nastavitev, se modalno okno strežnika lahko prikaže v ospredju ali pa nas pričaka slika, podobna spodnji:

Slika 18-13: Prikazano modalno okno strežnika



Izkaže se, da modalno okno strežnika ni povsem modalno za preizkusni program, torej za odjemalca. To lahko preizkusimo tako, da znova skušamo klikniti kamorkoli po površini odjemalčevega okna. Namesto da bi zaslišali sistemski zvok, ki ponazarja onemogočeni obrazec, se čez čas izpiše opozorilo:

Slika 18-14: Opozorilo pri modalnem prikazu okna iz zunajprocesnega strežnika



Opozorilo nam ponuja gumb **Switch To**..., s katerim lahko prikličemo modalno prikazano okno drugega procesa in ga zapremo, da bomo lahko nadaljevali delo z odjemalcem. Funkcionalno je torej okno strežnika res modalno: z odjemalcem ne moremo početi ničesar, dokler ne zapremo modalno prikazanega okna. Ker pa sta obe okni v različnih procesih, se lahko zgodi, da odjemalčevo pogovorno okno prekrije strežnikovega.

Sedaj si oglejmo še značilnosti prikaza nemodalnega okna. V odjemalčevem oknu kliknemo gumb z napisom »Nemodalen«. Obe prikazani okni sta popolnoma neodvisni drugo od drugega. Ker je okno strežnika v drugem procesu, si ga odjemalec ne more lastiti, niti ni njegova življenjska doba kakorkoli povezana z njim. To lahko preizkusimo tako, da končamo izvajanje testnega odjemalca. Nemodalno prikazano pogovorno okno strežnika bo še vedno prikazano, čeprav je mogoče, da je zakrito s kakim drugim oknom (prikažemo ga lahko z opravilno vrstico ali preklopom s kombinacijo tipk Alt+Tab).

Če pogovorno okno strežnika zapremo, ta ne bo končal svojega izvajanja. Preklopiti moramo spet v prvi izvod visual basica in nasilno prekiniti izvajanje strežnika. Tako vedenje preizkušanega strežnika je koristno pri iskanju napak, vendar se razlikuje od vedenja samostojnega strežnika v prevedeni obliki (datoteka EXE). V tem primeru se strežnik samodejno naloži in zažene ob zahtevi po njegovi storitvi in tudi samodejno odstrani, ko je sproščena zadnja referenca na katerega izmed njegovih predmetov.

Kaj smo se naučili iz prejšnjega zgleda?

- Modalna pogovorna okna zunajprocesnih strežnikov se ne pokažejo vedno v ospredju, čeprav preprečujejo nadaljnjo uporabo odjemalca.
- Nemodalna pogovorna okna so povsem neodvisna od odjemalca in imajo svojo življenjsko dobo. Naložen obrazec strežnika lahko vztraja pri življenju tudi po tem, ko ni več nobenega odjemalca, ki ga uporablja, kar preprečuje odstranitev strežnika iz pomnilnika.
- Skratka: zunajprocesni strežniki niso najprimernejši za prikaz pogovornih oken.

Posodabljanje strežnika

Dokler s funkcionalnostjo zunajprocesnega strežnika nismo povsem zadovoljni, ga bomo uporabljali in posodabljali znotraj posebnega izvoda visual basica. Pri posodabljanju kode že napovedanega strežnika moramo biti previdni in upoštevati postopek, ki ga bomo pravkar spoznali:

1. Najprej zaustavimo vse morebitne odjemalce, ki se izvajajo v drugih izvodih visual basica ali v katerem izmed programov, ki podpirajo programiranje avtomatizacije.
2. Pri vseh odjemalcih v visual basicu sedaj *izklopimo* uporabo razredne knjižnice strežnika. Izberemo ukaz **References**... iz glavnega menuja **Project** ter na seznamu izključimo knjižnico strežnika (ker so vključene knjižnice vedno pri vrhu seznama, je ne bo težko najti).
3. Vemo že, da se preizkušani strežnik vede malo drugače kakor njegova samostojna različica, prevedena v izvršljivo datoteko EXE, zato moramo tudi strežnik ročno zaustaviti. Preklopimo v njegov izvod visual basica in z ukazom **End** nasilno končamo njegovo izvajanje.
4. Sedaj lahko posodobimo njegovo kodo in morebitni uporabniški vmesnik.
5. Posodobljeni strežnik vedno poženemo z ukazom **Start With Full Compile** (ali z ukazom Start, če smo onemogočili prevajanje na zahtevo, postopek je opisan v zadnjem namigu). Če bi ga pognali z nepopolnim prevajanjem, nas med izvajanjem lahko preseneti napaka, ki je posledica skladenjske napake.

Predčasna zaustavitev strežnika zaradi izvajalne napake ima lahko hude posledice za nadaljevanje preizkušanja, saj se odjemalci lahko še vedno sklicujejo na izvode razredov, ki jih ni več (neveljavne reference).

6. Vrnemo se v izvod visual basica z odjemalcem in znova vključimo razredno knjižnico strežnika. Po potrebi še posodobimo odjemalca in ga poženemo.

NAMIG Neprestanemu izklapljanju in vklapljanju razredne knjižnice se izognemo tako, da omogočimo binarno združljivost različic. Več o njej bomo izvedeli iz splošnih napotkov v nadaljevanju.

Asinhrono obveščanje

Najuporabnejša zmogljivost zunajprocesnih strežnikov je asinhrono delovanje. Odjemalec lahko strežniku naloži časovno zahtevno opravilo in nadaljuje svoje delo. Strežnik ga lahko kadarkoli obvesti o koncu dolgotrajne operacije, ki ga odjemalec zazna prek mehanizma dogodkov. Ker lahko do takega dogodka pride kadarkoli med izvajanjem odjemalčeve kode, pravimo, da z njo ni usklajen (*sinhroniziran*), v žargonu rečeno, je *asinhron*.

Za zgled bomo pripravili strežnik, ki bo v vnaprej določenih časovnih intervalih preverjal pogoj in če bo resničen (pri nas bo to kar vedno), sprožil asinhrono obvestilo z javnim dogodkom. Tovrsten strežnik je uporaben npr. za preverjanje elektronskega poštnega nabiralnika ali nadziranje zunanjih naprav, ki so priključene na računalnik.

Razvoj novega zunajprocesnega strežnika začnemo z izbiro novega projekta vrste »ActiveX EXE«. Najprej določimo lastnosti projekta. Izberemo ukaz **Project ➪ Project1 Properties**...

- V vnosno polje **Project Name** vnesemo simbolično ime razredne knjižnice. Naj bo to v našem primeru »TestObvestil«.
- V vnosno polje **Project Description** vnesemo daljše, opisno ime strežnika. Naj bo to »Asinhrono obveščanje«.

Sedaj poimenujmo še obstoječi razredni modul. Privzeto ime »Class1« v oknu za nastavljanje lastnosti spremenimo v »Razred«. Lastnosti Instancing se zaenkrat ne dotikamo.

Za preverjanje pogoja v rednih časovnih intervalih bi nam prišel najbolj prav gradnik Timer. Tega lahko uporabimo samo na posebnih modulih – obrazcih. Zato z ukazom **Add Form** iz glavnega menuja **Project** dodamo v projekt obrazec uporabniškega vmesnika. Nanj namestimo gradnik Timer in nastavimo lastnosti po naslednji tabeli:

Gradnik	Lastnost	Nastavitev
Obrazec (*Form*)	(Name)	FrmTimer
Ura (*Timer*)	(Name)	Timer1
	Interval	5000

Gradnik Timer bo skrbel za proženje dogodka vsakih pet sekund. Celoten obrazec in vsi gradniki na njem so zasebna last strežnika, zato bomo gradnik izpostavili prek javnega vmesnika razreda Razred. Obrazec tako ne potrebuje nikakršne programske kode. Vrnemo se v okno s kodo razrednega modula »Razred« (najlaže s projektnim oknom) in vnesemo naslednjo programsko kodo:

```
Private frm As Form1
Private WithEvents tmr As Timer

Public Event Dogodek()

Private Sub Class_Initialize()
  Set frm = New FrmTimer
  Load frm
  Set tmr = frm.Timer1
End Sub

Private Sub Class_Terminate()
  Set tmr = Nothing
  Unload frm
  Set frm = Nothing
End Sub
```

Napovedali smo dve zasebni predmetni spremenljivki in en javni dogodek. Predmetni spremenljivki vzpostavimo in uničimo skupaj s predmetom. Ker se predmetna spremenljivka tmr sklicuje na gradnik uporabniškega vmesnika, je vrstni red stavkov nadvse pomemben.

Ker smo predmetno spremenljivko tmr napovedali z določilom WithEvents, lahko na seznamu elementov, levo na vrhu kodnega okna, izberemo predmet tmr in takoj bomo dobili telo napovedi odzivne procedure njegovega edinega dogodka, dogodka Timer. Dodamo mu naslednjo kodo:

```
Private Sub tmr_Timer()
  ' preverjanje pogoja, če je pogoj
  ' izpolnjen, potem ...
  RaiseEvent Dogodek
End Sub
```

Spet imamo pred seboj dobro znani zgled. Ker je obrazec zasebna last strežnika, moramo njegove zmožnosti izpostaviti z verižno povezanimi dogodki. V uporabnejšem zgledu bi pred proženjem dogodka poskrbeli za preverjanje določenih pogojev, npr. spremembe vsebine v elektronskem poštnem nabiralniku ali spremembi na vhodno/izhodnem vmesniku.

Strežnik je s tem nared. Poženemo ga z izbiro ukaza **Start With Full Compile** iz glavnega menuja **Run** ter nadaljujemo delo v drugem izvodu razvojnega okolja visual basica.

Odjemalec bo zelo preprost. Delo v drugem izvodu visual basica začnemo z novim projektom vrste »Standard EXE«. Preden zapišemo kako vrstico kode, moramo vklopiti novo razredno knjižnico pravkar razvitega strežnika. Na seznamu knjižnic poiščemo prej zapisano opisno ime strežnika (»Asinhrono obveščanje«) in vklopimo njegovo knjižnico. Potem je na voljo uporaba razreda »Razred«.

Na uporabniški vmesnik obrazca dodamo ukazni gumb in zapišemo naslednjo programsko kodo:

```
Private WithEvents r As Razred

Private Sub Command1_Click()
  Set r = New Razred
End Sub

Private Sub r_Dogodek()
  Debug.Print "Obvestilo!"
End Sub
```

Če projekt sedaj poženemo in kliknemo ukazni gumb, se bo približno vsakih pet sekund v takojšnjem oknu izpisalo sporočilo o prejetem dogodku.

Izvodi razreda

Poglejmo sedaj, kakšno je vzpostavljanje razreda »Razred«, ki ga priskrbi zunajprocesni strežnik. Uporabniški vmesnik odjemalca bomo dopolnili s še enim ukaznim gumbom in spremenili programsko kodo v naslednjo obliko:

```
Private WithEvents r1 As Razred
Private WithEvents r2 As Razred
Private sngDiff As Single

Private Sub Command1_Click()
  Set r1 = New Razred
End Sub

Private Sub Command2_Click()
  Set r2 = New Razred
End Sub

Private Sub r1_Dogodek()
  Debug.Print "Dogodek!"
  If sngDiff = 0 Then
    sngDiff = Timer
  Else
    Debug.Print "Razlika: " & CInt(Timer - sngDiff)
    sngDiff = Timer
  End If
End Sub
```

```
Private Sub r2_Dogodek()
  Debug.Print "Dogodek!"
  If sngDiff = 0 Then
    sngDiff = Timer
  Else
    Debug.Print "Razlika: " & CInt(Timer - sngDiff)
    sngDiff = Timer
  End If
End Sub
```

Odjemalec bo sedaj uporabil dva izvoda razreda Razred. Poženimo program in najprej kliknimo prvi ukazni gumb. Če bomo počakali nekaj časa, bomo ugotovili, da si dogodki še vedno sledijo v zaporedju pet sekund. Sedaj kliknimo še drugi ukazni gumb. Ugotovili bomo, da se očitno ustvarita dva izvoda razreda Razreda, saj si dogodka sledita v manjših časovnih intervalih. Gumbe lahko klikamo še naprej in tako spreminjamo razdelitev petsekundnega intervala[61].

Kaj bi se zgodilo, če bi preverjanje pogoja v strežniku zaklenilo določen vir, npr. vhodno/izhodni vmesnik, ki nadzira neko zunanjo napravo? Ker lahko ustvarimo več izvodov razreda, bi en izvod preprečil delo drugim, ki bi se pritožili z izvajalno napako, kar bi lahko imelo hude posledice. Kako bi lahko omogočili hkratno uporabo takega vira več odjemalcem?

Problem rešimo tako, da pri strežniku vpeljemo nov razred, ki nadzira priklop različnih odjemalcev (*connector class*) in ustvari ali uniči en sam potreben izvod razreda »Razred«, ki bo dostopen le v enem izvodu. Preden se lotimo spreminjanja strežnika, se spomnimo zgoraj opisanega postopka (glej Posodabljanje strežnika).

V izvodu visual basica, kjer se izvaja strežnik, zaustavimo izvajanje in posodobimo nastavitev lastnosti razrednega modula Razred. Njegovo lastnost Instancing, ki je doslej dopuščala delo z več izvodi, spremenimo v »2 – PublicNotCreatable«. Razred bo sedaj dostopen za rabo, vendar odjemalci ne bodo mogli izdelovati njegovih izvodov.

Za vzpostavljanje in uničevanje predmeta bo poskrbel poseben razred, ki pa potrebuje pomoč samostojnega modula s kodo. V projekt ga dodamo z izbiro ukaza **Add Module**... iz glavnega menuja **Project**. V galeriji izberemo običajni modul. V njegovi kodi bomo napovedali dve globalni spremenljivki:

```
Public gRazred As Razred
Public glngStevec As Long
```

Prva hrani referenco enega samega izvoda razreda Razred, druga pa šteje preklopljene odjemalce. Sedaj lahko v projekt dodamo nov razred. V glavnem menuju **Project** izberemo ukaz **Add Class Module**... in v galeriji običajni razredni modul. Ime novega razreda naj se glasi »Priklop«, njegova lastnost Instancing pa naj ostane na privzeti nastavitvi »5 – MultiUse«, saj je namen tega razreda, da ga odjemalci ustvarjajo v poljubnem številu izvodov.

---

[61] Seštevek obeh intervalov zaradi zaokroževanja s funkcijo CInt ni nujno vedno 5.

Programska koda novega razreda »Priklop« se glasi:

```
Public Property Get Razred() As Razred
  Set Razred = gRazred
End Property

Private Sub Class_Initialize()
  If gRazred Is Nothing Then Set gRazred = New Razred
  glngStevec = glngStevec + 1
End Sub

Private Sub Class_Terminate()
  glngStevec = glngStevec - 1
  If glngStevec = 0 Then Set gRazred = Nothing
End Sub
```

En sam vzpostavljen izvod razreda Razred bomo izpostavili kot lastnost razreda Priklop. V vzpostavitvi moramo poskrbeti za morebitno vzpostavitev prvega in edinega izvoda razreda Razred, pozneje pa samo povečujemo števec sklicevanj (*reference counter*), ki ga hrani globalna spremenljivka glngStevec. Ob uničenju posameznega izvoda razreda Priklop ta števec zmanjšamo, ko pade na nič, uničimo tudi vzpostavljeni izvod razreda Razred. Nov strežnik poženemo s popolnim prevajanjem.

Sedaj posodobimo kodo odjemalca v drugem izvodu visual basica. Po novem se glasi takole:

```
Private p1 As New Priklop
Private p2 As New Priklop
Private WithEvents r1 As Razred
Private WithEvents r2 As Razred
Private sngDiff As Single

Private Sub Command1_Click()
  Set r1 = p1.Razred
End Sub

Private Sub Command2_Click()
  Set r2 = p2.Razred
End Sub

Private Sub r1_Dogodek()
  Debug.Print "Dogodek!"
  If sngDiff = 0 Then
    sngDiff = Timer
  Else
    Debug.Print "Razlika: " & CInt(Timer - sngDiff)
    sngDiff = Timer
  End If
End Sub

Private Sub r2_Dogodek()
  Debug.Print "Dogodek!"
  If sngDiff = 0 Then
    sngDiff = Timer
```

```
  Else
    Debug.Print "Razlika: " & CInt(Timer - sngDiff)
    sngDiff = Timer
  End If
End Sub
```

Odzivni proceduri dogodkov sta ostali nespremenjeni, spremenilo se je le vzpostavljanje predmetov razreda Razred. Sedaj jih ne ustvarimo neposredno (tega nam strežnik niti ne dopušča), ampak jih preberemo kot lastnost predmeta razreda Priklop. Pred preizkusom odjemalca ne pozabimo na izklop in vnovični vklop spremenjene razredne knjižnice strežnika.

Sedaj lahko oba gumba klikamo v poljubnih presledkih, dogodka si vedno sledita nemudoma, drug za drugim, vmes pa mine okrog pet sekund. Očitno je, da oba predmeta posredno uporabljata enak izvod razreda Razred.

Prijatelji

Pri izdelavi strežnika smo morali poseči po globalnih spremenljivkah, saj ne poznamo drugega načina za komunikacijo med razredi. Pri strežnikih ActiveX lahko pri napovedi podprogramov uporabimo določilo **Friend**, ki označuje *prijateljske* podprograme. Prijateljski podprogram je na voljo v vseh drugih razredih projekta (torej strežnika), kakor da bi bil javen, ni pa na voljo v razredni knjižnici, zato ga odjemalci ne morejo uporabiti. Pri reševanju prejšnjega problema nam prijateljski podprogram ne more kaj dosti pomagati, zna pa priti prav pri kakršnikoli zasebni komunikaciji med posameznimi izvodi razredov strežnika.

Naslednji zgled prikazuje prijateljsko funkcijo, ki vrne stanje predmeta in tako omogoča pogovor med različnimi predmeti:

```
Friend Function VrniStanje() As RazredStanja
  VrniStanje = mStanje
End Function
```

Predlog Friend lahko začne napoved kateregakoli podprograma, tudi podprograma lastnosti:

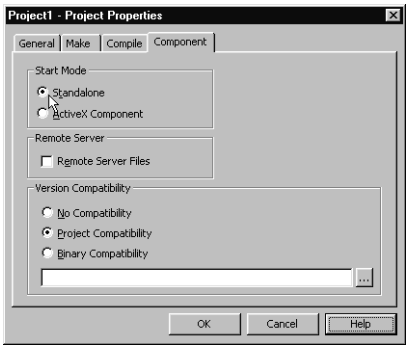
```
Public Property Get Lastnost() As String
  Lastnost = mLastnost
End Property

Friend Property Let Lastnost(NovaVrednost As String)
  mLastnost = NovaVrednost
End Property
```

Lastnost je znotraj projekta na voljo za branje in zapisovanje, navzven pa samo za branje.

Samostojni strežnik

Zunajprocesni strežniki se lahko izvajajo v ozadju kot gradniki ActiveX ali pa so samostojni programi, ki svoje storitve ponujajo drugim odjemalcem. Tako je izdelana večina sodobne namenske programske opreme za Okna, npr. Microsoft Excel. Uporabljamo ga lahko kot običajen izvršljiv program, hkrati pa je sposoben streči odjemalcem, ki znajo uporabiti avtomatizacijo. Če želimo, da se zunajprocesni strežnik zažene kot samostojni program, v glavnem menuju **Project** izberemo ukaz **Project Properties**... (ime ukaza je odvisno od imena dejavnega projekta). V prikazanem pogovornem oknu izberemo jeziček **Component** ter v skupini **Start Mode** vklopimo stikalo **Standalone**.

Slika 18-15: Določanje samostojnega zunajprocesnega strežnika



Strežniki, ki so samostojni programi, naj bi vsebovali uporabniški vmesnik. Vendar v nastavitvah zagonskega elementa ne moremo izbrati nobenega obrazca. Na voljo je le procedura Sub Main. Samostojni zunajprocesni strežniki zato morajo imeti poseben modul s kodo, v katerem zapišemo zagonsko proceduro. Njena oblika je ponavadi naslednja:

```
Sub Main()
  If App.StartMode = vbSModeStandalone Then
    Form1.Show
  End If
End Sub
```

Z lastnostjo **StartMode** vgrajenega predmeta App lahko ugotovimo, kako je operacijski sistem zagnal program. Če gre za samostojni strežnik, prikažemo prvi obrazec uporabniškega vmesnika.

> **NAMIG** Pri samostojnih programih, ki so hkrati tudi zunajprocesni strežniki ActiveX (npr. Microsoft Excel), lahko operacijski sistem program požene v ozadju. To se zgodi, če kateri izmed programov, ki podpirajo programirano avtomatizacijo (npr. sam visual basic), zahteva izvod izpostavljenega razreda.

Povratni klici

Samostojni zunajprocesni strežnik seveda lahko nastopa tudi v vlogi odjemalca, ki izkorišča storitve drugega zunajprocesnega strežnika. V tem primeru lahko za njuno medsebojno komunikacijo poleg dogodkov uporabimo tudi mehanizem *povratnih klicev*. Povratni klic (*callback*) omogoča strežniku, da izve naslov odjemalčevega podprograma, ki ga v ustreznem trenutku pokliče in s tem obvesti odjemalca o spremembi svojega stanja. Oglejmo si nekaj razlik med javnimi dogodki in povratnimi klici:

- Javni dogodek je kakor oddajanje sporočila v vse smeri, ne da bi vedeli, ali sporočilo sploh kdo posluša. Povratni klic pomeni predajo sporočila natančno poznanemu poslušalcu ali več poslušalcem.
- Vrstni red, v katerem poslušalci prejmejo sporočilo, je pri dogodkih nedoločen, pri uporabi povratnih klicev pa s programsko kodo določamo vrstni red naslavljanja poslušalcev. Zato je pri dogodkih strežnik onemogočen, dokler vsi odjemalci ne obdelajo sporočila, pri uporabi povratnih klicev pa strežnik prevzame nadzor po vsakem posameznem klicu.
- Posledica zgoraj opisane značilnosti je tudi omejenost pri spremljanju povratnih informacij odjemalcev pri več poslušalcih. Če dogodek opremimo s parametrom, ki se prenaša po referenci, bo ob povratku vidna le sprememba, ki jo je na parametru izvedel zadnji poslušalec. Pri uporabi povratnih klicev lahko analiziramo spremembo parametra za vsakega poslušalca posebej.
- Pri uporabi povratnih klicev lahko prestrežemo napake, ki se pripetijo pri poslušalcu znotraj povratno klicanega podprograma.

Povratne klice vzpostavimo z vmesnikom (*interface*). Strežnik mora pripraviti vmesnik, ki vključuje napovedi vseh povratno klicanih podprogramov. Če je teh povratno klicanih podprogramov več, jih lahko razporedimo v več vmesnikov in s tem razbremenimo programerja, ki bo zadolžen za njihovo izpeljavo. Po pravilih poimenovanja, ki jih uporabljajo gradniki ActiveX, naj bi se ime vmesnika začelo z veliko črko I (*interface*). Oglejmo si, kako bi lahko strežnik, uporabljen v prejšnjih zgledih, opremili s preprostim vmesnikom za povratne klice:

```
' koda vmesnika v javnem razrednem modulu IObvesti

Public Sub Koncano()

End Sub
```

Vsi poslušalci bodo morali izdelati javen razred, ki bo izpeljal vmesnik IObvesti. Vzpostavljanje javnih razrednih modulov, ki vsebujejo vmesnike, določimo z lastnostjo Instancing, nastavljeno na »2 – PublicNotCreatable«.

Strežnik je zadolžen še za pripravo funkcij, ki bodo omogočale priklop poslušalcev. Te funkcije morajo prejeti parameter z izvodom razreda, ki izpeljuje prej določeni

vmesnik. Za naš zgled bi lahko javni razred Priklop opremili z naslednjo javno metodo:

```
Private mPoslusalec As IObvesti

Public Sub Naroci(ByVal Poslusalec As IObvesti)
  Set mPoslusalec = Poslusalec
End Sub
```

Posredovani izvod razreda, ki izpeljuje zahtevani vmesnik, shranimo v zasebno predmetno spremenljivko razreda. Zagotovo pozna metodo Koncano, ki jo lahko pokličemo, kadarkoli želimo odjemalca obvestiti o dogodku.

Oglejmo si še naloge odjemalca. Ta mora vsebovati poseben javni razredni modul, ki izpeljuje potrebni vmesnik za povratni klic. Prav to je vzrok, da so povratni klici mogoči le pri pogovoru med strežniki ActiveX, saj običajni projekti ne poznajo javnih razrednih modulov. Njegovo lastnost Instancing prav tako nastavimo na »2 – PublicNotCreatable«, saj bomo potrebne izvode ustvarili v samem odjemalcu. Oglejmo si zgled novega javnega razreda odjemalca, denimo, da se imenuje Prijava:

```
' Koda javnega razrednega modula Prijava

Implements IObvesti

Private Sub IObvesti_Koncano()
  Debug.Print "Dogodek!"
End Sub
```

Novi javni razred izpelje edino metodo vmesnika IObvesti.

Sedaj je vse pripravljeno za priklop poslušalca. Izvedemo ga tako, da ustvarimo javni izvod razreda Prijava in ga posredujemo naročniški metodi strežnika:

```
Public gPriklop As Priklop

Set gPriklop = New Priklop
gPriklop.Naroci gPrijava
```

Strežnik sedaj natančno pozna svojega poslušalca in ga lahko obvesti o spremembi stanja s klicem javne metode razreda Priklop, ki je izpeljal vmesnik IObvesti. Če želi strežnik obveščati več poslušalcev, mora vzpostaviti podatkovno strukturo – npr. dinamično polje ali, še bolje, zbirko – ki hrani reference za vse prijavljene poslušalce.

Vidimo, da je uporaba povratnih klicev nekoliko bolj zapletena, vendar ponuja mnogo prednosti.

NAMIG Celotno programsko kodo zgleda povratnega klica najdete na priloženi plošči CD-ROM v mapi \Primeri\4-Razredi\2-Gradniki\2-ActiveX EXE\3-Povratni klici\.

Napotki

Oglejmo si še nekaj splošnih napotkov, ki jih velja upoštevati pri razvoju strežnikov ActiveX. Za začetek si bomo ogledali enolično označevanje razrednih knjižnic, ki je ključ za nadzor različic.

GUID

V sistemskem registru so različne predmetne knjižnice in vmesniki enolično označeni s 128-bitno oznako, imenovano GUID (*Globally Unique Identifier*) ali tudi UUID (*Universally Unique Identifier*). Algoritem, ki se uporablja pri izračunavanju teh oznak, zagotavlja, da se enaka oznaka ne bo ponovila nikjer drugje na svetu.

Združljivost različic

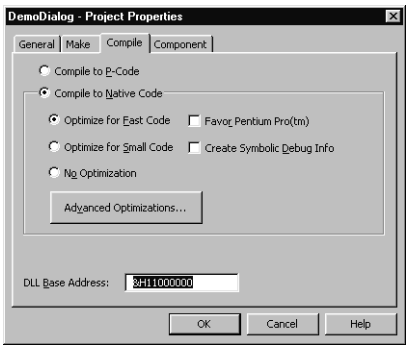
Kadar uporabnik izkoristi storitve izpostavljenih strežnikov, se njihove enolične oznake zapišejo v prevedeno izvršljivo datoteko odjemalca. Kadarkoli bo prevedeni program želel znova uporabiti storitev strežnika, bo operacijski sistem preveril register in v njem iskal ustrezno, prej shranjeno oznako. Če je ne najde, bo sporočil napako in odjemalec ne bo deloval pravilno.

Med razvojem novih gradnikov visual basic izračuna novo enolično oznako vedno, ko prevedemo projekt. Tako vedenje je primerno med razvojem, lahko pa povzroči težave, če želimo končani izdelek nadgrajevati z novimi različicami. Ko je razvoj za prvo izdajo končan, lahko visual basicu naročimo, naj od tega trenutka naprej vsako prevedeno različico projekta opremi z enako oznako. To bo omogočilo uporabnikom, ki bodo staro različico strežnika nadgradili z novo, da še naprej uporabljajo njene storitve. V tem primeru je silno pomembno, da visual basicu naročimo, naj samodejno povečuje oznako različice.

Prevajanje projekta z enako enolično oznako dosežemo tako, da pri nastavitvah projekta vključimo stikalo **Binary Version Compatibility**[62]. Zaporedje ukazov, ki nas pripelje do te nastavitve, se glasi: **Project** ➪ **Project properties**... ➪ jeziček **Component** ➪ skupina **Version Compatibilty**. Sedaj je treba vključiti tudi samodejno povečevanje oznake različice. To dosežemo s preklopom na jeziček **Make** in vklopom stikala **Auto Increment** v skupini **Version Number**. S tem smo določili *binarno združljivost* različic strežnika.

Ko vklopimo binarno združljivost različic, bo visual basic tudi pazil, da ne spremenimo že določenih vmesnikov. Lahko dodajamo nove razrede in spreminjamo kodo znotraj metod, na vsako spremembo javnih članov razreda in parametrov metod pa bo odgovoril z opozorilom o nezdružljivosti različic. Če se kljub temu odločimo, da bomo prevedli nezdružljivo različico strežnika, visual basic spremeni glavno oznako različice in predlaga prevajanje v datoteko z novim

[62] Tovrstno združljivost lahko vklopimo šele po vsaj enem uspešnem prevajanju projekta.



Slika 18-16: Vklop binarne združljivosti različic

imenom. S tem preprečimo prekrivanje starejše različice v ciljnih računalnikih, ki že uporabljajo starejšo različico strežnika.

> **NAMIG** Preverjanje različic je že vgrajeno v namestitveni postopek, ki ga izdela orodje Application Setup Wizard.

Izhodiščni naslov

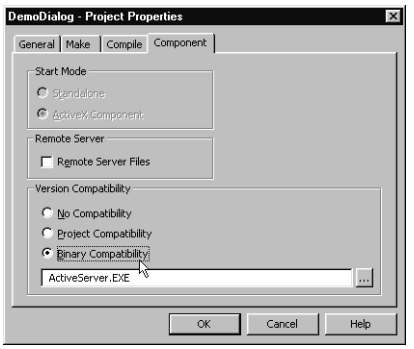
Pri uporabi več znotrajprocesnih gradnikov pri istem projektu na učinkovitost precej vpliva dobro izbran *izhodiščni naslov*. Izhodiščni naslov (*base address*) je pomnilniški naslov v navideznem pomnilniku[63] (*virtual memory*), v katerega se naloži prevedena koda znotrajprocesnega gradnika. Če je ta naslov že zaseden s kodo samega programa ali drugega znotrajprocesnega strežnika, je potrebno prestavljanje (*rebasing*), ki je časovno precej potratno.

Ker je izhodiščni naslov za vse znotrajprocesne strežnike nastavljen na isto vrednost, dokler ga ročno ne spremenimo, moramo prestavljati vedno, kadar pri istem projektu uporabimo dva gradnika, razvita z visual basicom. To je lahko kar precejšnja težava, če se zavedamo, da so tudi vsi zunanji gradniki ActiveX (knjižnice OCX) pravzaprav znotrajprocesni strežniki.

Izhodiščni naslov določimo v nastavitvah projekta, ki jih dosežemo z ukazom **Project Properties**... iz glavnega menuja **Project**. Pod jezičkom **Compile** bomo našli vnosno polje **DLL Base Address**, v katero lahko vnesemo nov izhodiščni naslov. Na začetku je vnesen v šestnajstiški obliki, lahko pa ga vnesemo tudi z desetiško zapisanim številom. Na voljo imamo navidezne naslove med 4 MB in 2 GB[64].

[63] Navidezni pomnilnik omogoča preslikavo logičnih naslovov v fizične. Operacijski sistem se lahko pretvarja, da ima na voljo ogromne količine pomnilnika, posebni algoritmi, podprti tudi v strojni opremi (mikroprocesorju), pa omogočajo učinkovito preslikavo naslovov v precej manjši fizični pomnilnik in včasih tudi v zunanji pomnilnik na disku.

[64] 32-bitni procesi se v operacijskih sistemih Windows izvajajo v naslovnem prostoru 4 GB, pri čemer so spodnji 4 MB in zgornja 2 GB rezervirana za sam operacijski sistem.



Slika 18-17: Določanje izhodiščnega naslova

Spodnja meja ni najboljši izbor, saj se na tem naslovu praviloma naloži koda samega procesa. Priporočajo izbor v mejah od 16 MB do 2 GB (manj velikost samega znotrajprocesnega strežnika). Naslov mora biti večkratnik 64 KB, ker je to enota strani pomnilnika (*memory page*). Ker ne moremo vedeti, kam se bo nalagala prevedena koda drugih znotrajprocesnih gradnikov, je ponavadi najbolje, če uporabimo generator naključnih števil (s čim daljšo periodo), ki izbere poljuben naslov znotraj prej omenjenega intervala in ga zaokroži navzgor na prvi večkratnik 64 KB.

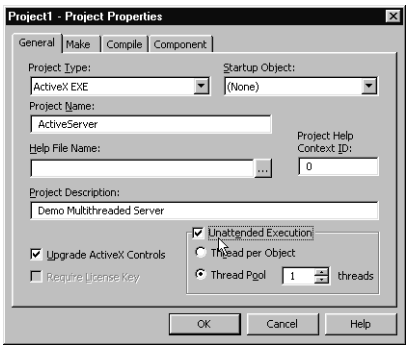
Večnitnost

V visual basicu je mogoče pri strežnikih, ki v ničemer ne komunicirajo z uporabnikom, torej se lahko izvajajo brez posebne *pozornosti* uporabnika (*unattended execution*), omogočiti *večnitno* izvajanje (*multithreading*). Ko vklopimo tovrstno izvajanje, hkrati preprečimo prikaz kateregakoli pogovornega okna, tudi tistega, ki ga prikaže vgrajena funkcija MsgBox. Pri tovrstnih strežnikih je zato priporočljivo uporabiti zapis v dnevnik, ki ga omogoča metoda vgrajenega predmeta App (uporabo dnevnika smo obravnavali v drugem delu knjige).

Večnitnost je posebej pomembna pri strežnikih, pri katerih lahko obdelava ene metode vzame veliko časa, medtem ko neka druga metoda zahteva le drobec časa. Zaradi enonitnosti lahko klic kratke metode, če pride prepozno, prepreči nadaljnje izvajanje odjemalca, vse dokler se strežnik ne sprosti po obdelavi dolgotrajne metode.

Večnitnost omogočimo z vklopom stikala **Unattended Execution** v nastavitvah projekta. Stikalo je na voljo le pri projektih, ki ne vključujejo uporabniškega vmesnika, in hkrati onemogoči prikaz pogovornih oken, ki so izid kake vgrajene funkcije.

Slika 18-18: Omogočena večnitnost



Strežniki v strežniku

Najboljša različica visual basica (Enterprise Edition) omogoča pripravo oddaljenih zunajprocesnih strežnikov (»ActiveX EXE«). Oddaljeni strežniki lahko ponujajo svoje storitve tudi prek omrežja. Namestimo jih v poseben računalnik, vključen v omrežje, svoje storitve pa ponujajo navzven prek *oddaljene avtomatizacije* (*remote automation*) ali protokola DCOM (*Distributed Component Object Model*).

Omogočanje tovrstnih strežnikov je v visual basicu čisto preprosto: v nastavitvah projekta pod jezičkom **Component** vklopimo stikalo **Remote Server Files**. Ob prevajanju bo visual basic pripravil posebno datoteko s podaljškom VBR, ki jo zna uporabiti pripomoček za izdelavo namestitvenega postopka Application Setup Wizard.

Slika 18-19: Omogočanje oddaljene strežbe

Gradniki vmesnika

Posebna oblika znotrajprocesnih strežnikov ActiveX so zunanji gradniki uporabniškega vmesnika. Uporabni so v visual basicu, pa tudi v vseh drugih orodjih, ki podpirajo zunanje gradnike ActiveX (npr. Borland Delphi). Nov gradnik uporabniškega vmesnika lahko dodamo vsaki obliki projekta v visual basicu,

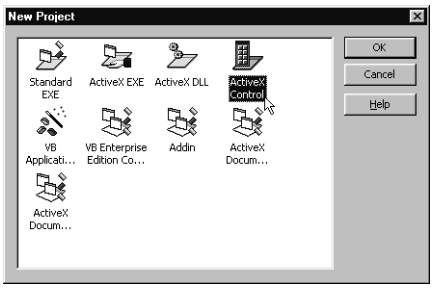
podobno kot razredni modul. Seveda se razlika pokaže pri vnovični rabi izdelanega gradnika. Če je gradnik del zaprtega projekta, ga ne moremo uporabiti nikjer drugje. Gradnike je zato bolje razviti v posebni obliki projekta, izid katere je prevedena datoteka v obliki OCX. To datoteko je mogoče vključiti v razvojno okolje kot vsak zunanji gradnik.

Pred razvojem novega gradnika, ki ga želimo izkoristiti v drugih projektih, je nujno dobro načrtovanje. Potek razvoja novega gradnika uporabniškega vmesnika bi lahko razčlenili na naslednje korake:

- Ugotavljanje zmogljivosti, ki jih želimo ponuditi prek gradnika.
- Izbira izhodišča razvoja. Na voljo imamo razvoj povsem novega gradnika, posodobitev obstoječega gradnika ali izdelavo gradnika, ki združuje funkcionalnost več obstoječih gradnikov.
- Preizkušanje in dopolnjevanje strežnika izvajamo z drugim, testnim projektom znotraj razvojnega okolja visual basica.
- Načrtovanje vizualnega vmesnika, torej videza gradnika. Dodajanje morebitnih obstoječih gradnikov na vmesnik novega gradnika.
- Načrtovanje programskega vmesnika ter kodiranje razreda, ki bo določal nov gradnik. Pisanje morebitne pomožne programske kode.
- Če gradnik vsebuje posebne lastnosti, mu lahko dodamo tudi nova okna za določanje njihovih vrednosti (*property pages*).
- Prevajanje v knjižnico OCX in preizkus z vsemi ciljnimi razvojnimi orodji. Med drugim ga lahko preizkusimo v novem izvodu visual basica, kamor ga vključimo kot nov zunanji gradnik.
- Morebitna priprava namestitvene različice.

Spoznavanje

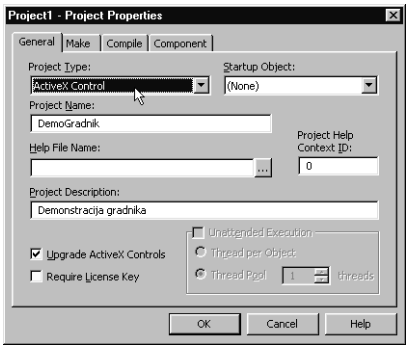
Nov gradnik uporabniškega vmesnika začnemo razvijati s posebno obliko projekta. V galeriji različnih vrst projektov, ki se prikaže po izbiri ukaza **New Project**, iz glavnega menuja **File** izberemo nov projekt vrste »ActiveX Control«.



Slika 18-20: Izbira novega projekta vrste »ActiveX Control«

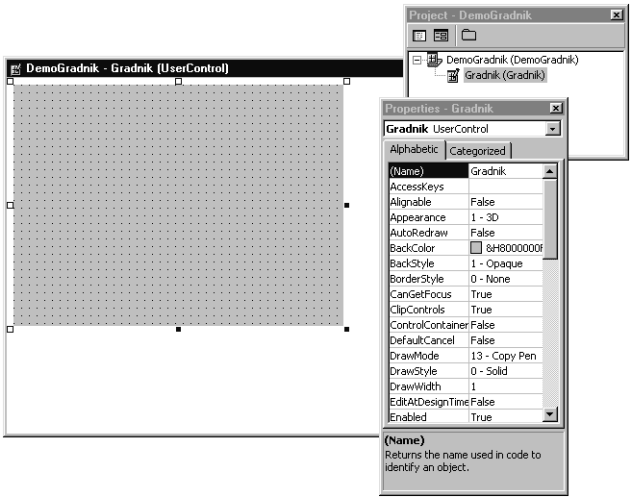
Novo odprti projekt na prvi pogled spominja na značilne projekte vrste »Standard EXE«. Uporabniški vmesnik novega gradnika urejamo v urejevalniku (*designer*) vrste UserControl, ki precej spominja na vgrajeni urejevalnik obrazcev. Tudi pri izdelavi novega gradnika se izplača nastaviti simbolično in opisno ime projekta, ki ju dosežemo z ukazom **Project1 Properties**... iz glavnega menuja **Project**. Za naš zgled nastavimo simbolično ime na »DemoGradnik«, opisno ime pa na »Demonstracija gradnika«.

Slika 18-21: Nastavljanje lastnosti projekta novega gradnika



Urejevalniku gradnika pripada okno za nastavljanje lastnosti, v katerem najdemo precej novih, ki jih doslej še nismo zasledili pri nobenem pripravljenem gradniku. Pomagale nam bodo pri razvoju. Kot pri vseh drugih elementih, je tudi tu najpomembnejša lastnost simbolično ime novega gradnika, ki ga določa lastnost Name. Mi jo nastavimo na preprosto ime »Gradnik«. Nov gradnik je pravzaprav posebna oblika razrednega modula, zato to ime hkrati predstavlja razred, ki mu bodo posamezni izvodi novo razvitega gradnika pripadali.

Slika 18-22: Videz pomembnejših oken razvojnega okolja pri izdelavi novega gradnika

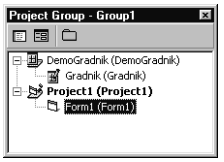


Urejevalnik omogoča nastavitev privzete velikosti novega gradnika. V tako določenem obsegu se bo gradnik izrisal, če ga vmesniku obrazca dodamo z dvakratnim klikom njegove ikone v orodjarni. Ponavadi privzeta velikost ni najprimernejša, zato gradnik nekoliko pomanjšamo.

Preizkusni projekt

Projekt vrste »ActiveX Control« je pravzaprav posebna oblika znotrajprocesnega strežnika, ki pa ga ne moremo zagnati. Za preizkus gradnika moramo v skupino dodati nov, običajni projekt. V glavnem menuju **File** izberemo ukaz **Add Project**... (pozor, *Add* in ne New ali Open) ter v galeriji običajni projekt vrste »Standard EXE«. V projektnem oknu opazimo, da je novo dodani projekt takoj označen kot zagonski (njegovo ime je izpisano krepko).

Slika 18-23: Projektno okno s skupino projektov: razvojnim in preizkusnim



Gradniki uporabniškega vmesnika poznajo dve, delno prekrivajoči se in delno različni vrsti vedenja:

- Vedenje med načrtovanjem vmesnika.
- Vedenje med izvajanjem.

Zunanji gradniki »živijo« že med načrtovanjem, kar pomeni, da se bo del programske kode gradnika izvajal, tudi ne da bi preizkusni projekt zagnali. Razvojno okolje visual basica ponuja preprosto in učinkovito metodo, s katero lahko med načrtovanjem preklapljamo z razvoja na uporabo novega gradnika.

Za prikaz te metode bomo novemu gradniku z zaenkrat še praznim uporabniškim vmesnikom dodali nekaj programske kode. Obdelali bomo odziv na dogodek Resize, ki se pripeti ob spremembi velikosti gradnika, povsem enako kakor pri drugih gradnikih. V projektnem oknu izberemo nov gradnik in v urejevalniku dvakrat kliknemo njegovo površino. S tem prikažemo okno s kodo, ki pripada novemu gradniku. Po privzeti nastavitvi se znajdemo znotraj odziva na dogodek Initialize, zato z desnega seznama dogodkov izberemo **Resize**. Zapišimo naslednjo programsko kodo:

```
Private Sub UserControl_Resize()
Static i As Integer
  i = i + 1
  Debug.Print i & ". nova velikost."
End Sub
```

Odzivna procedura šteje vse povzročene dogodke in izpisuje kratko pojasnilo v takojšnje okno. Kakor vidimo, pripada novemu gradniku v imenu odzivne procedure predlog UserControl, podobno kakor vsakemu obrazcu v njegovem kodnem oknu pripada predlog Form.

Če bi šlo za navaden projekt v visual basicu, bi ga lahko pognali, pri načrtovanju gradnika pa je drugače. Vse, kar moramo storiti, je, da zapremo okno urejevalnika vmesnika novega gradnika (kodno okno lahko ostane prikazano).

> **NAMIG** Če smo v oknu s kodo gradnika, lahko hitro poženemo kodo s kombinacijo tipk Shift+F7, ki ji sledi Ctrl+F4.

Ozrimo se sedaj po orodjarni. V njej je simbolična ikona novega gradnika!

Slika 18-24: Nov gradnik je že na voljo v orodjarni

Sedaj lahko gradnik dodamo uporabniškemu vmesniku preizkusnega projekta. Preklopimo vanj in prikažimo obrazec Form1. Z dvakratnim klikom gumba v orodjarni, ki predstavlja naš novi gradnik, ga dodamo vmesniku v privzeti velikosti. V takojšnjem oknu lahko preberemo:

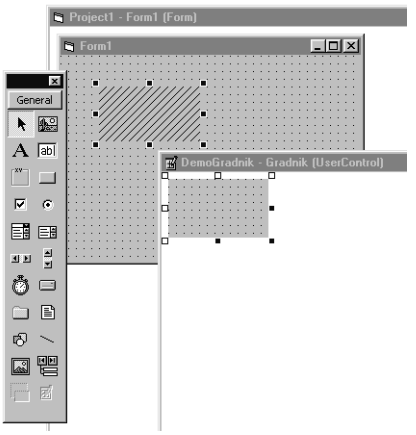
```
1. nova velikost.
```

> **NAMIG** Če takojšnje okno morda ni vidno, ga prikažemo z ukazom Immediate Window iz glavnega menuja View.

V dokaz, da se koda novega gradnika že izvaja, spremenimo njegovo velikost v vmesniku. Po vsaki spremembi se v takojšnjem oknu prikaže novo obvestilo. Če preklopimo v okno obrazca s kodo, ugotovimo, da vsak uporabniški gradnik brez posebnih nastavitev že pozna dogodke DragDrop, DragOver, GotFocus in LostFocus. Seveda lahko omogočamo tudi druge dogodke in celo dodajamo nove.

Če znova prikličemo urejevalnik vmesnika gradnika, se ustrezen gumb v orodjarni onemogoči, izrisani gradnik na vmesniku obrazca pa postane šrafiran, saj se koda

sedaj ne izvaja več in videz ne more odražati dejanskega vedenja gradnika. Brž ko urejevalnik znova skrijemo, se podoba gradnika obnovi.



Slika 18-25: Nov gradnik je onemogočen, če ga urejamo.

Če spreminjamo programsko *kodo* gradnika in je urejevalnik skrit, je gradnik omogočen, vendar ne odraža sprememb. Zato je v priročnem menuju obrazca na voljo nov ukaz, **Update UserControls**, ki vnovič vzpostavi posodobljeni gradnik.

> **NAMIG** Visual basic uporablja nekoliko nespreten algoritem, ki ugotavlja spremembe v kodi, zato se včasih po spremembi v kodi gradnika in izdanem ukazu Update Controls ne odzove najbolje. Težavo rešimo tako, da testni projekt poženemo s popolnim prevajanjem. Ko končamo izvajanje testnega projekta, bo tudi v razvojnem okolju vse delovalo v skladu z novo kodo.
>
> Tudi pri razvijanju novih gradnikov uporabniškega vmesnika velja, da je projekte najbolje prevajati s popolnim prevajanjem. Bodisi izberemo ustrezen ukaz iz menuja Run ali pa onemogočimo prevajanje na zahtevo (*compile on demand*).

Življenjski krog

Pri vseh gradnikih smo spoznavali njihov življenjski krog, ki ga označujejo posebni dogodki. Za obrazec so to Initialize, Load, QueryUnload in Unload, za razredni modul pa Initialize in Terminate. Posebne dogodke prejme tudi gradnik uporabniškega vmesnika. Zanj je pravilna uporaba teh dogodkov ključna, saj mora dosledno obnavljati svoje stanje med oblikovanjem in izvajanjem programa in tudi med različnimi sejami visual basica.

Ključni dogodki v življenju gradnika so:

- **Initialize**. Vzpostavitev razreda, kakor pri vseh vrstah razrednih modulov.
- **InitProperties**. Vzpostavljanje lastnosti, ki določajo videz gradnika.
- **ReadProperties**. Branje shranjenih lastnosti.
- **WriteProperties**. Shranjevanje nastavljenih lastnosti.
- **Terminate**. Konec uporabe razreda, podobno kakor pri drugih razrednih modulih.

> **NAMIG** Če še naprej primerjamo uporabniški gradnik vmesnika z obrazcem, bi lahko rekli, da sta dogodka Load in Unload zamenjana z dogodkoma ReadProperties in WriteProperties.

Vsako izmed odzivnih procedur teh dogodkov bomo opremili z izpisom pojasnila v takojšnje okno. Tako bomo natančno izvedeli, kdaj se kateri dogodek uporabi:

```
Private Sub UserControl_Initialize()
  Debug.Print "dogodek Initialize"
End Sub

Private Sub UserControl_InitProperties()
  Debug.Print "dogodek InitProperties"
End Sub

Private Sub UserControl_ReadProperties(PropBag As PropertyBag)
  Debug.Print "dogodek ReadProperties"
End Sub

Private Sub UserControl_WriteProperties(PropBag As PropertyBag)
  Debug.Print "dogodek WriteProperties"
End Sub

Private Sub UserControl_Terminate()
  Debug.Print "dogodek Terminate"
End Sub
```

Od prej nam je ostala tudi odzivna procedura dogodka Resize, ki je nikar ne odstranimo iz kode.

Če sedaj gradnik spravimo v dejavno stanje (s skrivanjem urejevalnika), lahko v takojšnjem oknu zagledamo naslednja sporočila:

```
dogodek Initialize
dogodek InitProperties
1. nova velikost
```

Začudeni? Saj nismo dodali novega gradnika, ampak le spremenili kodo obstoječega. Gradnike moramo opazovati z dveh stališč:

- Uporabnik ima pred seboj pripravljen gradnik, od katerega pričakuje določene usluge. Zanj je ves čas na obrazcu en sam gradnik, ki se z nastavitvami lastnosti različno vede.
- Razvijalec gradnika ve, da je precej popravil programsko kodo, zato je moral visual basic uničiti prejšnji izvod gradnika in vzpostaviti novega, ki odraža spremembe v kodi. Enak učinek dosežemo tudi z izdajo ukaza **Update UserControls** iz priročnega menuja obrazca.

Zakaj potem nismo videli izpisa obvestila o dogodku Terminate? Zato ker prvi izvod gradnika, ki se je uničil, še ni imel določenega odziva na ta dogodek.

Spremenimo položaj oziroma velikost gradnika. Poglejmo, kaj se bo zgodilo z vsebino takojšnjega okna, ko testni projekt še poženemo.

```
dogodek Initialize
dogodek InitProperties
1. nova velikost
dogodek WriteProperties
dogodek Terminate
dogodek Initialize
dogodek ReadProperties
1. nova velikost
```

Prva dva dogodka, ki sledita ob zagonu testnega projekta, sta WriteProperties in Terminate. Označujeta uničenje izvoda gradnika, ki smo ga uporabljali med načrtovanjem. Ob nastajanju testnega obrazca se vzpostavi nov gradnik, zato dogodka Initialize in ReadProperties. Zaradi zaporedja dogodkov WriteProperties in ReadProperties lahko gradnik med izvajanjem prevzame vse lastnosti gradnika, pripravljenega med načrtovanjem programa.

> **NAMIG** Dogodek WriteProperties se pripeti, če smo med *načrtovanjem* spremenili katero od lastnosti gradnika, npr. velikost.

Ko končamo izvajanje testnega projekta, bo uničenje izvajalnega izvoda gradnika pospremil samo dogodek Terminate. Natančno tako, iz izkušenj že vemo, da se spremembe lastnosti gradnika, ki jih opravimo med izvajanjem programa, ne odrazijo na gradniku, ko se vrnemo k načrtovanju. Izvajalni izvod gradnika zato ne bo nikoli prejel dogodka WriteProperties, ta dogodek je namenjen izključno načrtovanemu izvodu gradnika.

Če je vsebina takojšnjega okna že nepregledna, lahko označimo vse besedilo v njem in ga odstranimo s pritiskom na tipko Delete. Pregledno takojšnje okno potrebujemo za ponazoritev še ene značilnosti iz življenjskega kroga gradnikov uporabniškega vmesnika. Na obrazec testnega projekta dodajmo nov izvod našega gradnika. Znova lahko zagledamo zaporedje dogodkov:

```
dogodek Initialize
dogodek InitProperties
1. nova velikost
```

Podobno kakor pri drugih razredih se za vsak izvod gradnika ustvari nov predmet, torej nov izvod razreda, ki prejme dogodka Initialize in InitProperties. Odzivno proceduro dogodka InitProperties uporabimo za nastavitev privzetih lastnosti gradnika in za morebitne druge reakcije ob vzpostavitvi novega izvoda gradnika.

Opazujmo še, kaj se zgodi, ko zapremo okno urejevalnika obrazca. V takojšnjem oknu se prikaže par sporočil WriteProperties in Terminate.

```
dogodek WriteProperties
dogodek WriteProperties
dogodek Terminate
dogodek Terminate
```

Vsak izvod gradnika je dobil priložnost, da shrani svoje lastnosti, preden se uniči. Sedaj pa znova prikličimo prej zaprto okno urejevalnika obrazca. To najlaže storimo tako, da dvakrat kliknemo njegovo ikono v projektnem oknu. Takojšnje okno pokaže:

```
dogodek Initialize
dogodek ReadProperties
1. nova velikost
dogodek Initialize
dogodek ReadProperties
1. nova velikost
```

Gradnika sta prejela dogodek Initialize, ne pa tudi dogodka InitProperties, saj sta bila na obrazec nameščena že prej. Torej za nastavljanje privzetih lastnosti gradnika ne bomo uporabili odziva na dogodek Initialize. Ta dogodek je namenjen drugim vzpostavitvenim opravilom.

NAMIG Programsko kodo zgledov najdete na priloženi plošči CD-ROM v mapi \Primeri\4-Razredi\2-Gradniki\3-ActiveX Control\1-Spoznavanje\.

Izdelovanje gradnikov

Pri uvodu v spoznavanje lastnih gradnikov uporabniškega vmesnika smo že omenili tri načine izgradnje novega gradnika. To so:

- Ustvarjanje novega gradnika. V tem primeru moramo sami poskrbeti za izris grafične podobe gradnika v vseh stanjih, ki jih želimo obravnavati (npr. običajno, izbrano, vključeno, izključeno itn.).
- Posodabljanje obstoječega gradnika. Z dodatno kodo lahko obstoječemu gradniku dodamo nove lastnosti in dogodke, spremenimo privzete nastavitve

in izpostavimo nove elemente vmesnika. A tako precej težko bistveno spremenimo vizualno podobo gradnika.

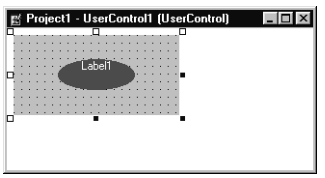
- Združevanje obstoječih gradnikov v nov uporabniški gradnik. To je podobno prejšnjemu in nam omogoča, da funkcionalno združujemo sorodne gradnike, npr. oznako in vnosno polje.

Posodabljanje združitve

Za začetek si bomo ogledali pripravo gradnika, ki bo združil in posodobil vedenje dveh obstoječih gradnikov. Nadaljevali bomo s projektom, ki smo ga uporabili že pri demonstraciji prikaza življenjskega kroga uporabniškega gradnika. Naš novi gradnik bosta sestavljala oznaka in lik (gradnik Shape), predstavljal pa bo izboljšano oznako.

Za začetek dela si pripravimo obstoječe gradnike. Te, ki sestavljajo nov, uporabniški gradnik, imenujemo tudi *temeljni* gradniki (*constituent controls*). Najprej dodajmo gradnik lik, zatem pa še oznako, in jima nastavimo naslednje lastnosti:

Gradnik	Lastnost	Nastavitev
Lik (*Shape*)	(Name)	shpLik
	BorderStyle	0 – Transparent
	FillColor	&HFF (rdeča)
	FillStyle	0 – Solid
	Shape	2 – Oval
Oznaka (*Label*)	(Name)	lblOznaka
	Alignment	2 – Center
	AutoSize	True
	BackStyle	0 – Transparent
	ForeColor	&HFFFFFF (bela)



Slika 18-26: Temeljni gradniki novega gradnika v njegovem urejevalniku

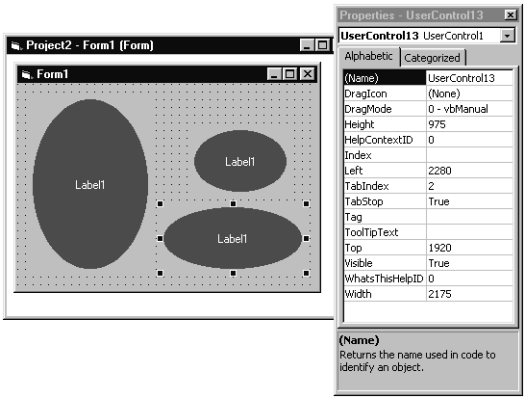
Sedaj se lahko lotimo kode novega gradnika. Kot že vemo, je za gradnik nadvse pomemben dogodek Resize. V njegovi odzivni proceduri moramo uskladiti velikost temeljnih gradnikov z velikostjo gradnika na vmesniku, kot jo je izbral uporabnik. V našem zgledu je ta koda naslednja:

```
Private Sub UserControl_Resize()
  shpLik.Move 0, 0, ScaleWidth, ScaleHeight
  lblOznaka.Move 0, (ScaleHeight - lblOznaka.Height) / 2, ScaleWidth
End Sub
```

Obseg koordinat, ki so nam na voljo na vidni površini uporabniškega gradnika, je vedno določen z intervalom 0 ... ScaleWidth po širini in 0 ... ScaleHeight po višini. Skala so vedno *twipi*, ne glede na uporabljeno skalo vsebnika, v katerem je gradnik.

Sedaj lahko skrijemo urejevalnik gradnika in preizkusimo vedenje novega gradnika na vmesniku testnega projekta. Opazimo, da se lik vedno raztegne čez vso razpoložljivo površino gradnika, besedilo v oznaki pa je izpisano na sredini. Še pogled v okno za nastavljanje lastnosti. Na voljo so splošne lastnosti gradnika, nikjer pa ni opaziti posebnih lastnosti temeljnih gradnikov, kot sta npr. Caption ali Shape. Če na obrazec namestimo več izvodov gradnika, opazimo, da imajo vsi enak napis: »Label1«. To ponazarja tudi spodnja slika:

Slika 18-27: Uporaba novega gradnika na vmesniku



Naslednja naloga, ki nas čaka, je torej izpostavljanje obstoječih lastnosti temeljnih gradnikov. Ker je nov gradnik le posebna oblika razrednega modula (in znotraj procesnega strežnika), dodamo lastnost povsem običajno, s parom podprogramov. Pomagamo si lahko z ukazom **Add Procedure** iz menuja **Tools**:

```
Public Property Get Caption() As String
  Caption = lblOznaka.Caption
End Property

Public Property Let Caption(ByVal NewCaption As String)
  lblOznaka.Caption = NewCaption
  PropertyChanged "Caption"
End Property
```

Pazimo, da bomo lastnosti priredili poseben podatkovni tip. Potem bo koda hitrejša, uporabnik gradnika pa bo natančneje vedel, kaj določena lastnost lahko sprejme. Pomembno: posebnost podprograma Property Let je klic vgrajenega podprograma **PropertyChanged**. V vseh podprogramih, ki spreminjajo lastnost gradnika, moramo

poklicati ta podprogram, ki mu kot parameter posredujemo niz z imenom lastnosti. Klic podprograma PropertyChanged osveži vsebino okna za nastavljanje lastnosti ter opozori razvojno okolje, da se je določena lastnost spremenila.

Sedaj poskrbimo še za začetno vrednost lastnosti Caption. Omenili smo že, da je za nastavitev privzete vrednosti katerekoli lastnosti edino primeren kraj odzivna procedura dogodka InitProperties:

```
Private Sub UserControl_InitProperties()
  Caption = Extender.Name
End Sub
```

Vsebino oznake preberemo kar iz imena posebnega vgrajenega predmeta, imenovanega **Extender**. Ta predmet vključuje lastnosti gradnika, ki jih ne more nadzorovati sam, ampak jih nadzoruje njegov *vsebnik*. Te lastnosti so Left, Height, Visible, Tag in tudi privzeto ime gradnika, lastnost Name. Seznam vseh vsebniških lastnosti najdemo v oknu za nastavljanje lastnosti pri lastnem gradniku, ki še ne vsebuje programske kode (glej zadnjo sliko). Kaj bi se zgodilo, če bi določili novo, lastno lastnost in bi jo prav tako poimenovali Name? Nič posebnega, to lastnost bi lahko uporabljali le interno, znotraj kode gradnika, navzven pa bi bila vedno vidna lastnost Name, kot jo določa vsebnik oziroma vgrajeni predmet Extender.

> **NAMIG** Vse lastnosti predmeta Extender, razen lastnosti Visible, so na voljo samo za branje. Paziti moramo še na eno, zelo pomembno dejstvo: okolja, ki omogočajo uporabo gradnikov ActiveX, ne podpirajo nujno vseh lastnosti predmeta Extender. Pri najbolj trdoživih gradnikih je pred poseganjem k njihovim lastnostim koristno vklopiti prestrezanje izvajalnih napak.

V našem zgledu posnemamo vedenje običajne oznake, ki za svoj privzeti napis prav tako uporabi kar svoje ime. Pri načrtovanju lastnih gradnikov je dobro, če čimbolj posnemamo vedenje obstoječih gradnikov, saj s tem olajšamo rabo.

Pri vzpostavljanju lastnosti nam pride prav še en vgrajen predmet, predmet **Ambient**. Njegove lastnosti ponazarjajo predlagani videz novega gradnika, vključno z uporabljenimi barvami, pisavo, krajevno oznako itn. Predlagani videz je večinoma določen z vsebnikom gradnika, če pa vsebnik določene lastnosti ne pozna, visual basic priskrbi privzeto nastavitev. Če npr. želimo, da bi oznaka uporabila enako pisavo kakor njen vsebnik, dopolnimo podprogram v naslednjo obliko:

```
Private Sub UserControl_InitProperties()
  Caption = Extender.Name
  lblOznaka.Font = Ambient.Font
End Sub
```

Če želimo usklajenost vzdrževati tudi pri nadaljnji uporabi gradnika, zapišemo še odzivno proceduro za dogodek **AmbientChanged**. Koristno je, če lastnemu gradniku

določimo posebno lastnost (poimenujemo jo npr. SyncAmbient), ki določa, ali se podoba gradnika usklajuje z vsebnikom ali ne.

Najpomembnejša lastnost predmeta Ambient je **UserMode**, ki določa, v kateri fazi je gradnik. Če je postavljena na resnično vrednost, je gradnik v izvajanju, drugače pa je pri načrtovanju vmesnika.

Sedaj smo poskrbeli, da je prek okna za nastavljanje lastnosti dostopna lastnost Caption, ki se prek vnesene programske kode preslika v istoimensko lastnost temeljne oznake. Naredimo preprost preizkus: med načrtovanjem spremenimo napis v neko novo vrednost in poženimo testni projekt. Hmm, ko program zaženemo, gradnik znova izpiše privzeto nastavitev? Spomnimo se življenjskega kroga. Izvod, ki smo ga uporabili med načrtovanjem, je že uničen, med izvajanjem se samodejno ustvari drug izvod, ki želi prebrati shranjene nastavitve. Aha, za odraz nastavitev med izvajanjem moramo zapisati kodo odzivnih procedur dogodkov WriteProperties in ReadProperties:

```
Private Sub UserControl_WriteProperties(PropBag As PropertyBag)
  PropBag.WriteProperty "Caption", Caption, Extender.Name
End Sub

Private Sub UserControl_ReadProperties(PropBag As PropertyBag)
  Caption = PropBag.ReadProperty("Caption", Extender.Name)
End Sub
```

Zaradi uporabe posredovanega predmeta razreda **PropertyBag** (»vreča lastnosti«) je zapisovanje in branje posamezne lastnosti povsem preprosto. Zgolj uporabimo pripravljeno metodo **WriteProperty** ali **ReadProperty**. Pri zapisu lastnosti posredujemo tri parametre: niz z imenom lastnosti, njeno nastavitev in njeno privzeto vrednost. Privzeto vrednost zapišemo zato, da je visual basic takrat, ko se *ni* spremenila, sploh ne bo shranil. S tem znatno izboljšamo čitljivost shranjene oblike obrazca (glej dodatek B). Funkcija za branje lastnosti prejme dva parametra: niz z imenom lastnosti in privzeto vrednost, ki se uporabi, če lastnost ni posebej shranjena. Paziti moramo, da se privzete vrednosti za posamezno lastnost, uporabljene v metodah WriteProperty in ReadProperty ter v odzivu na dogodek InitProperties, ujemajo.

NAMIG V oknu za nastavljanje lastnosti je mogoče pri vsaki lastnosti prebrati tudi krajši opis. No, naša izpostavljena lastnost Caption tega pojasnila nima. Dodamo ga povsem enako, kakor dodamo pojasnilo vsakemu članu razreda: z ukazom **Procedure Attributes**... iz glavnega menuja **Tools**. Z zaprtega seznama izberemo ime lastnosti in v okvir Description vnesemo opis izbrane lastnosti.

Pri načrtovanju gradnikov se pojavi vprašanje, kako poimenovati lastnosti in v katerem jeziku vnašati njihove opise. No, odgovor je predvsem odvisen od ciljne publike. Če razvijamo nov gradnik, namenjen širšemu trgu, je verjetno najbolje uporabiti angleščino.

Po drugi strani pa je slovenščina verjetno boljša odločitev, če pripravljamo gradnik za zaprto skupino uporabnikov, npr. za interno rabo znotraj podjetja.

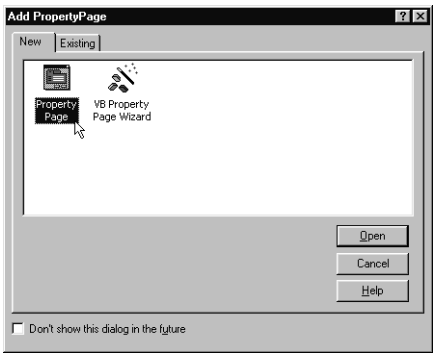
Strani lastnosti

Lastnosti, ki pripadajo specifičnim podatkovnim tipom ali naštevnemu tipu, bodo samodejno prikazane v oknu za nastavljanje lastnosti. Nekatere posebne lastnosti, kot je barva, pisava ali bitna slika, nastavljamo s posebnimi orodji. Zunanji gradniki pa poznajo še en način nastavljanja lastnosti. S simbolično lastnostjo »(Custom)« lahko prikličemo pogovorno okno lastnosti (*property pages*), v katerem so pregledno zbrane značilne lastnosti izbranega gradnika, ponavadi na več straneh. To pogovorno okno je še posebej uporabno pri bolj zapletenih gradnikih, za uspešno rabo katerih je nujno nastaviti več povezanih lastnosti. Brez posebnega pogovornega okna za nastavljanje lastnosti ne gre takrat, ko gradnike, shranjene v knjižnici OCX, uporabimo tudi v drugih razvojnih okoljih, ki ne podpirajo posebnega okna za nastavljanje lastnosti.

Posamezne *strani* za nastavljanje lastnosti lahko načrtujemo tudi za lastne gradnike. Vsaka stran je v pogovornem oknu predstavljena s svojim jezičkom. Visual basic samodejno skrbi za njihov preklop in sproži ustrezne dogodke ob izbiri gumbov Apply, OK in Cancel. Tudi naš posodobljeni gradnik bomo popestrili z lastno *stranjo lastnosti* (*property page*). Preden projektu dodamo novo stran, se prepričajmo, da je v projektnem oknu izbran projekt z gradnikom in ne morda testni projekt. Strani lastnosti je namreč mogoče, podobno kakor tudi nov uporabniški gradnik, dodati v vsako vrsto projekta.

Za novo stran lastnosti v glavnem menuju **Project** izberemo ukaz **Add Property Page**... Prikaže se galerija, v kateri izberemo običajno, prazno okno.

Slika 18-28: Izbiranje nove strani lastnosti



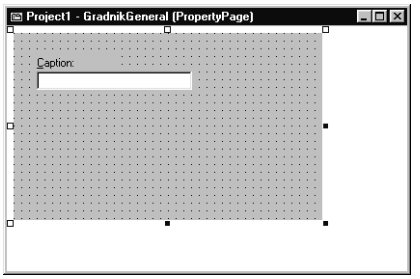
Tudi načrtovanje strani lastnosti spominja na načrtovanje obrazca. Najprej nastavimo poglavitno lastnost: ime. V oknu za nastavljanje lastnosti v prvo vrstico

vnesemo novo vrednost lastnosti (Name), naj bo to »GradnikGeneral«. S tem je določeno simbolično ime strani, ki bo uporabno v programski kodi. Vnesti moramo tudi opisno ime, ki bo vsebovano v jezičku te strani v pogovornem oknu lastnosti. Opisno ime določa lastnost Caption. Nastavimo jo na »General«. Zakaj nismo tudi simboličnega imena zapisali bolj preprosto kot »General«? Ker bomo lahko razvili več gradnikov in bo vsak imel splošno stran. Predlog imena naj zato ponazarja ime gradnika.

Stran lastnosti bomo v našem preprostem primeru uporabili le za nastavitev lastnosti Caption. Zato bomo na njen vmesnik dodali dva gradnika, oznako in vnosno polje, ki jima nastavimo naslednje lastnosti:

Gradnik	Lastnost	Nastavitev
Oznaka (*Label*)	(Name)	Label1
	Caption	&Caption:
	TabIndex	0
Vnosno polje (TextBox)	(Name)	txtCaption
	Text	(prazno)
	TabIndex	1

Videz vmesnika je razviden tudi s spodnje slike:



Slika 18-29: Vmesnik strani lastnosti

Sedaj se lahko posvetimo kodi, ki bo pripadala strani lastnosti. Odpremo njeno kodno okno (npr. z dvakratnim klikom njene površine) in izberemo odziv na dogodek **SelectionChanged**. Vnesemo naslednjo kodo:

```
Private Sub PropertyPage_SelectionChanged()
  txtCaption.Text = SelectedControls(0).Caption
End Sub
```

V odzivu na ta dogodek moramo uskladiti vsebino strani lastnost z *izbranimi* gradniki. Kot vemo, je na uporabniškem vmesniku obrazca mogoče hkrati izbrati tudi več gradnikov. Vsi izbrani gradniki so ponazorjeni z zbirko gradnikov

z imenom **SelectedControls**. Ker je okno za nastavljanje lastnosti prikazano nemodalno, se dogodek lahko pripeti kadarkoli, velikost polja SelectedControls pa se lahko ustrezno temu spremeni. Odločili smo se, da bomo vsebino vnosnega polja vedno uskladili s prvim izbranim gradnikom (ta zbirka označi prvi element z zaporedno številko 0).

Odziv na več izbranih gradnikov je v celoti prepuščen nam. Lahko se odločimo, da nekaterih lastnosti enostavno ne dovolimo spreminjati, če je izbran več kot en gradnik (zagotovo je taka lastnost ime gradnika – Name – ki mora biti enolična za vsak gradnik). Tako kakor pri vsaki zbirki dobimo število elementov v njej z branjem lastnosti Count. Naslednji zgled onemogoči izmišljeno vnosno polje, če je izbranih več gradnikov:

```
If SelectedControls.Count > 1 Then
  txtVnosLastnost.Enabled = False
Else
  txtVnosLastnost.Enabled = True
End If
```

Zdaj moramo poskrbeti še za usklajevanje v nasprotni smeri. Kadar uporabnik vnese novo vsebino na stran lastnosti in klikne gumba Apply ali OK, oziroma kadar zamenja stran v več strani dolgem pogovornem oknu, se pripeti dogodek **ApplyChanges**:

```
Private Sub PropertyPage_ApplyChanges()
Dim obj As Object
  For Each obj In SelectedControls
    If TypeName(obj) = "Gradnik" Then obj.Caption = txtCaption.Text
  Next
End Sub
```

V odzivno proceduro smo vnesli manjši varnostni mehanizem, ki preverja ime razreda, ki mu gradnik pripada. Ta mehanizem ni nujno potreben. Vsako pripravljeno stran lastnosti moramo ročno priključiti izbranim gradnikom v projektu, ki ga razvijamo. Če smo pri tem opravilu dovolj pazljivi in strani ne priključimo gradnikom, ki ne poznajo lastnosti Caption, bo visual basic sam poskrbel, da stran lastnosti ne bo vidna, če bo v izbrani skupini vsaj en gradnik, ki te lastnosti ne pozna. No, pri našem zgledu ne bo treba posebej paziti in bomo lahko razvito stran priključili kateremukoli gradniku v projektu. Razvite strani lastnosti lahko uporabimo le za gradnike v našem projektu.

Preostane nam še ena odzivna procedura. Gumb Apply omogoča takojšnjo uskladitev gradnika s spremenjeno vrednostjo lastnosti. Da bo visual basic vedel, kdaj naj bo gumb Apply omogočen, ga moramo o tem obvestiti. Pri nas bo za to opravilo najboljši odziv na dogodek Change samega vnosnega polja:

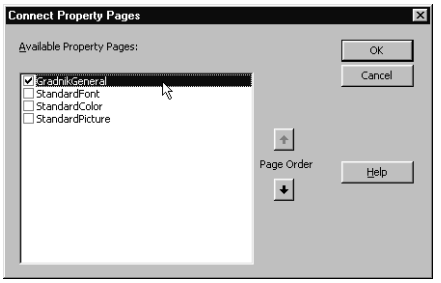
```
Private Sub txtCaption_Change()
  Changed = True
End Sub
```

Vse strani lastnosti poznajo vgrajeno lastnost Changed, ki naj ponazarja spremembe vrednosti lastnosti. Kadar ima lastnost resnično vrednost, je gumb Apply omogočen.

Razvoj strani lastnosti je sedaj končan, preostane nam še, da jo priklopimo na ustrezen gradnik. Pred priklopom postavimo stran v izvajanje. To storimo enako kakor pri gradniku, preprosto zapremo okno njenega urejevalnika.

Za priklop strani lastnosti najprej v projektnem oknu izberemo ikono gradnika in prikažemo njegov urejevalnik, Okno za nastavljanje lastnosti sedaj vsebuje lastnosti gradnika in med njimi najdemo tudi lastnost **PropertyPages**. Odpremo okno za njeno nastavitev in prikazal se bo seznam vseh strani lastnosti, ki so na voljo:

Slika 18-30: Seznam razpoložljivih strani lastnosti



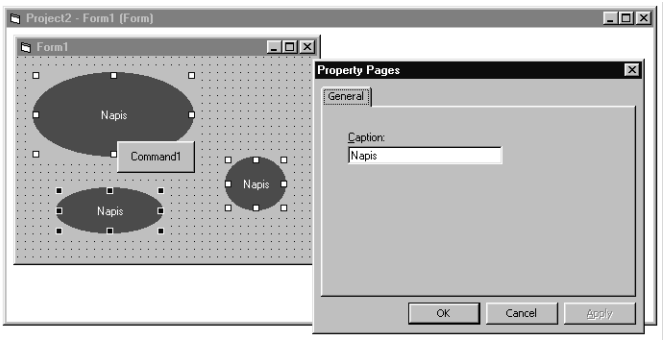
Na seznamu opazimo nekaj strani lastnosti, ki so vedno na voljo. To so standardne strani za urejanje barve, pisave in bitne slike. Za naš gradnik ne potrebujemo nobene izmed njih, temveč vklopimo samo prej pripravljeno stran s simboličnim imenom GradnikGeneral.

Sedaj lahko urejevalnik gradnika znova zapremo in se preselimo k vmesniku preizkusnega projekta. Na njegov obrazec dodajmo dva ali več lastnih gradnikov in kak vgrajeni gradnik, denimo ukazni gumb. Pripravljeno stran lastnosti lahko pokličemo na dva načina:

- V oknu za nastavljanje lastnosti izberemo simbolično lastnost »(Custom)«.
- V priročnem menuju lastnega gradnika izberemo ukaz **Properties**.

Prikaže se popolnoma oblikovano pogovorno okno, skupaj z jezičkom, tremi ukaznimi gumbi in vsebino prej pripravljene strani. Vsebina v vnosnem polju že odraža napis na gradniku. Vnesimo novo vrednost in kliknimo gumb **Apply**. Sprememba se odrazi na gradniku. S tipko Ctrl na vmesniku izberimo več lastnih gradnikov. Če spet poizkusimo spremeniti napis, se bo hkrati spremenil na vseh izbranih gradnikih. Vključimo v skupino še kak drug gradnik (npr. ukazni gumb, če smo ga prej dodali). Pogovorno okno za nastavljanje lastnosti bo izpisalo opozorilo, da ni izbranih gradnikov, ki bi jim skupaj odgovarjala katera izmed strani.

Slika 18-31: Uporaba strani lastnosti

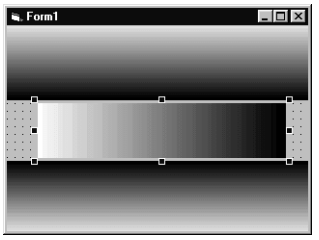


NAMIG Programsko kodo zgleda, dopolnjeno še z nekaj elementi, najdete na priloženi plošči CD-ROM v mapi \Primeri\4-Razredi\2-Gradniki\3-ActiveX Control\2-Izdelava\1-Gradnik\.

Preliv

Preden si ogledamo bolj splošne napotke za razvoj lastnih gradnikov, spoznajmo še način izdelave lastnega gradnika brez uporabe temeljnih gradnikov. Pri tovrstnih lastnih gradnikih imamo pri vizualni podobi gradnika popolno svobodo. To značilnost bomo koristno uporabili pri gradniku, ki bo omogočal zapolnitev poljubne površine s prelivom iz ene barve v drugo. Na spodnji sliki vidimo nekaj zgledov rabe:

Slika 18-32: Uporaba gradnika, ki ga bomo razvili.



Pri lastnih gradnikih, katerih vizualno podobo nadzorujemo sami, je ključnega pomena kodiranje odziva na dogodek Paint. Ta dogodek se pokliče vedno, kadar je treba osvežiti podobo gradnika.

Delo bomo začeli z odpiranjem novega projekta vrste »ActiveX Control«. Pričaka nas urejevalnik novega gradnika. Vse, kar storimo z njim, je, da nastavimo privzeto velikost gradnika. Preostanek dela bomo opravili v oknu s kodo.

Najprej si oglejmo del splošnih napovedi.

```
Public Enum GradFillType
  gfLeftRight
  gfTopBottom
End Enum

Private mColor1 As Long
Private mColor2 As Long
Private mFillType As GradFillType

Private mColor3 As Long
Private mStartRGB(0 To 2) As Byte
Private mDiffRGB(0 To 2) As Long
```

Za lažje kodiranje, predvsem pa za lažjo uporabo gradnika v drugih projektih najprej določimo naštevni tip. Ta tip določa dva različna načina zapolnitve s prelivom z leve proti desni ali od zgoraj navzdol. Sledijo zasebne spremenljivke, ki bodo hranile nastavitev posameznih lastnosti, nekaj spremenljivk pa potrebujemo za delo.

Sledi kodiranje lastnosti:

```
Public VisiClick As Boolean

Public Property Get FillType() As GradFillType
  FillType = mFillType
End Property

Public Property Let FillType(NewType As GradFillType)
  If NewType <> mFillType Then
    Select Case NewType
      Case gfLeftRight, gfTopBottom
        mFillType = NewType
        Refresh
      Case Else
        Err.Raise Number:=vbObjectError + 5001, _
          Description:="Invalid gradient fill type"
    End Select
  End If
End Property

Public Property Get Color1() As OLE_COLOR
  Color1 = mColor1
End Property

Public Property Let Color1(ByVal NewColor As OLE_COLOR)
  If NewColor <> mColor1 Then
    mColor1 = NewColor
    ResetPaintValues
    Refresh
  End If
End Property

Public Property Get Color2() As OLE_COLOR
  Color2 = mColor2
End Property
```

```
Public Property Let Color2(ByVal NewColor As OLE_COLOR)
  If NewColor <> mColor2 Then
    mColor2 = NewColor
    ResetPaintValues
    Refresh
  End If
End Property
```

Lastnost VisiClick določa, ali se gradnik vidno odzove na klik z miško. Ker nam za to lastnost zadošča poseben tip Boolean, nam ni treba zapisovati posebnih podprogramov. Drugače je z lastnostjo FillType, ki določa način zapolnitve. Posredovano vrednost moramo natančno analizirati, preden jo priredimo gradniku. Znotraj procedure opazimo še uporabo tehnike medpomnjenja (*caching*). S spreminjanjem vrednosti lastnosti se ukvarjamo le, če je nova vrednost različna od že shranjene. Ob uspešni spremembi lastnosti pokličemo metodo **Refresh**. Ta poskrbi za obnovitev vsebine gradnika, torej posredno povzroči dogodek Paint. Zelo pomembno: ob spremembi stanja, ki vpliva na vizualno podobo gradnika, nikoli ne pokličemo odzivne procedure dogodka Paint neposredno. Za to rabi metoda Refresh.

Podobno tehniko uporabimo tudi pri procedurah, ki omogočata uporabo lastnosti Color1 in Color2. Pri njiju opazimo še uporabo posebnega podatkovnega tipa, **OLE_COLOR**. Ta tip posamezni lastnosti dodeli posebni urejevalnik barv (povsem enak tistemu, ki se uporablja npr. za nastavitev lastnosti obrazca BackColor). Visual basic ponuja še nekaj takih tipov. Zbrani so v spodnji tabeli:

Tip	Enakovreden tipu	Opis
OLE_COLOR	Long	Omogoča izpostavljanje lastnosti, ki predstavlja barvo.
OLE_TRISTATE	Naštevni tip (0,1,2)	Omogoča izpostavljanje stanja običajnega stikala, ki je lahko vključeno, izključeno ali v nedoločenem stanju.
OLE_OPTEXCLUSIVE	Boolean	Omogoča izpostavljanje soodvisne vrednosti, kot jo pozna skupina opcijskih stikal v istem vsebniku. V skupini gradnikov v istem vsebniku ima lahko tako lastnost postavljeno na resnično vrednost samo eden.
OLE_CANCELBOOL	Boolean	Ta podatkovni tip lahko predpišemo parametru, ki ga posredujemo skozi dogodek. Če uporabnik tak parameter nastavi na resnično vrednost, bo preklical nadaljnje izvajanje dogodka (kot parameter Cancel v dogodku obrazca QueryUnload).

Pri vsaki spremembi barve ali vrste zapolnjevanja pokličemo posebno pomožno proceduro, ki vzpostavi parametre zapolnjevanja. Te pozneje uporabimo v odzivu na dogodek Paint, ki je zaradi nekaj vnaprejšnjega računanja hitrejši. Vsebina tega podprograma je naslednja:

```
Private Sub ResetPaintValues()
  mStartRGB(0) = GetR(Color2RGB(mColor1))
  mStartRGB(1) = GetG(Color2RGB(mColor1))
  mStartRGB(2) = GetB(Color2RGB(mColor1))

  mDiffRGB(0) = GetR(Color2RGB(mColor2)) - CInt(mStartRGB(0))
  mDiffRGB(1) = GetG(Color2RGB(mColor2)) - CInt(mStartRGB(1))
  mDiffRGB(2) = GetB(Color2RGB(mColor2)) - CInt(mStartRGB(2))
End Sub
```

Pravzaprav ne naredi drugega, kakor da izloči rdečo, zeleno in modro sestavino barve ter izračuna razliko med obema barvama, ki jo bo moral preliv postopoma premostiti. Tu smo uporabili še nekaj pomožnih funkcij, ki so zbrane v posebnem modulu s kodo, saj jih lahko uporabimo večkrat:

```
' Samostojen modul s kodo MdlColor
Declare Function GetSysColor Lib "user32" (ByVal nIndex As Long) As Long
Declare Function MulDiv Lib "kernel32" (ByVal nNumber As Long, _
  ByVal nNumerator As Long, ByVal nDenominator As Long) As Long

Public Function Color2RGB(ByVal c As Long) As Long
  If c < 0 Then
    Color2RGB = GetSysColor(c And &HFF)
  Else
    Color2RGB = c
  End If
End Function

Public Function GetR(ByVal c As Long) As Byte
  GetR = c And &HFF
End Function

Public Function GetG(ByVal c As Long) As Byte
  GetG = (c \ &H100) And &HFF
End Function

Public Function GetB(ByVal c As Long) As Byte
  GetB = (c \ &H10000) And &HFF
End Function
```

V modulu s kodo smo napovedali dve funkciji Windows API, ki nam bosta pomagali v nadaljevanju. Funkcija Color2RGB spremeni 32-bitni zapis barve, ki vključuje tudi paleto[65], v navadno, 24-bitno obliko RGB. Druge funkcije z bitnimi operacijami izluščijo posamezne barvne sestavine iz posredovane barve.

Vrnimo se v kodo novega lastnega gradnika. Oglejmo si ključno odzivno proceduro, ki upodobi vsebino gradnika:

[65] Kodiranje barv smo spoznali v drugem delu knjige, pri obravnavi grafičnih metod.

```
Private Sub UserControl_Paint()
Dim i As Integer, X As Integer
Dim R As Byte, G As Byte, B As Byte
Dim cLeft As Single, cRight As Single
Dim cTop As Single, cBottom As Single

  Select Case mFillType
    Case gfLeftRight
      cTop = 0
      cBottom = ScaleHeight
      X = ScaleWidth
    Case gfTopBottom
      cLeft = 0
      cRight = ScaleWidth
      X = ScaleHeight
  End Select

  For i = 0 To 50
    Select Case mFillType
      Case gfLeftRight
        cLeft = MulDiv(i, X, 50)
        cRight = MulDiv(i + 1, X, 50)
      Case gfTopBottom
        cTop = MulDiv(i, X, 50)
        cBottom = MulDiv(i + 1, X, 50)
    End Select
    R = (mStartRGB(0) + MulDiv(i, mDiffRGB(0), 50 - 1)) And &HFF
    G = (mStartRGB(1) + MulDiv(i, mDiffRGB(1), 50 - 1)) And &HFF
    B = (mStartRGB(2) + MulDiv(i, mDiffRGB(2), 50 - 1)) And &HFF
    Line (cLeft, cTop)-(cRight, cBottom), RGB(R, G, B), BF
  Next
End Sub
```

V zanki se sprehodimo po barvni razliki, ki je razdeljena na 50 segmentov. Vmesne barve izračunamo s funkcijo API, imenovano MulDiv, ki najprej zmnoži prva dva parametra in ju deli s tretjim. Zapolnitev segmenta dosežemo z uporabo metode Line s končnim parametrom BF.

Sedaj si lahko ogledamo, kako se bomo odzvali na klik z miško. Ustrezen dogodek je že na voljo, dodati moramo le kodo, ki ustrezno spremeni vizualno podobo gradnika, če do klika pride. Ne bomo uporabili kar odziva na dogodek klik, ampak bomo v proceduri MouseDown določili »vključeno« stanje, v odzivu na dogodek MouseUp pa obnovili prvotno stanje:

```
Private Sub UserControl_MouseDown(Button As Integer, Shift As Integer, _
X As Single, Y As Single)
  If VisiClick Then
    mColor3 = mColor1
    mColor1 = mColor2
    mColor2 = mColor3
    ResetPaintValues
    Refresh
  End If
End Sub
```

```
Private Sub UserControl_MouseUp(Button As Integer, Shift As Integer, _
X As Single, Y As Single)
  If VisiClick Then
    mColor3 = mColor1
    mColor1 = mColor2
    mColor2 = mColor3
    ResetPaintValues
    Refresh
  End If
End Sub
```

Vse skupaj je zelo preprosto. Če je vizualni odziv na klik sploh vključen, preprosto zamenjamo barvi in osvežimo prikaz gradnika. Pri vseh gradnikih ne gre tako preprosto. Ponavadi izrisovanje v različnih stanjih rešimo tako, da odziv na dogodek Paint izvedemo s stavkom Select Case, ki določa različna izrisovanja za različna stanja. V gradnik uvedemo posebno zasebno spremenljivko, ki v vsakem trenutku ponazarja trenutno stanje gradnika. Teh stanj je lahko kar nekaj, npr. običajno, vključeno, onemogočeno, z vnosno točko itn.

Oglejmo si še metode, ki skrbijo za pravilno vzpostavitev in shranjevanje nastavitev lastnosti:

```
Private Sub UserControl_InitProperties()
  Color1 = vbWhite
  Color2 = vbBlack
  FillType = gfLeftRight
  ResetPaintValues
  VisiClick = True
End Sub

Private Sub UserControl_WriteProperties(PropBag As PropertyBag)
  PropBag.WriteProperty "Color1", Color1, vbWhite
  PropBag.WriteProperty "Color2", Color2, vbBlack
  PropBag.WriteProperty "FillType", FillType, gfLeftRight
  PropBag.WriteProperty "VisiClick", VisiClick, True
End Sub

Private Sub UserControl_ReadProperties(PropBag As PropertyBag)
  Color1 = PropBag.ReadProperty("Color1", vbWhite)
  Color2 = PropBag.ReadProperty("Color2", vbBlack)
  FillType = PropBag.ReadProperty("FillType", gfLeftRight)
  ResetPaintValues
  VisiClick = PropBag.ReadProperty("VisiClick", True)
End Sub
```

No, nič novega, le lastnosti je nekaj več.

Sedaj pa si oglejmo še eno vznemirljivo možnost pri izdelovanju lastnih gradnikov: proženje dogodkov. Pri dogodkih imamo na voljo več možnosti:

- Lahko neposredno preslikamo obstoječi dogodek nad uporabniškim gradnikom ali nad enim izmed temeljnih gradnikov, uporabljenih pri načrtovanju novega gradnika.

- Obstoječi dogodek lahko analiziramo in prožimo naprej le v posebnem primeru.
- Ustvarimo popolnoma nov dogodek.

Seveda je delo z dogodki povsem enako kakor pri običajnih razrednih modulih. Naš zgled bomo posodobili tako, da bo gradnik ob kliku po svoji površini sprožil dogodek ColorSelected, ki bo s svojim parametrom ponazoril izbrano barvo.

Del splošnih napovedi dopolnimo z novo zasebno spremenljivko in napovedjo novega dogodka:

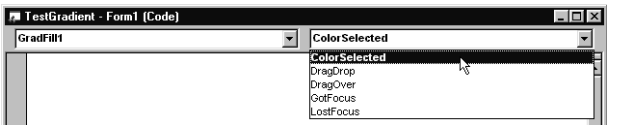
```
Private mSelColor As Long
Public Event ColorSelected(Color As Long)
```

Predelavo zahtevata še odziva na dogodka MouseDown in MouseUp:

```
Private Sub UserControl_MouseDown(Button As Integer, Shift As Integer, _
X As Single, Y As Single)
  mSelColor = Point(X, Y)

  ' ...

End Sub

Private Sub UserControl_MouseUp(Button As Integer, Shift As Integer, _
X As Single, Y As Single)

  ' ...

  RaiseEvent ColorSelected(mSelColor)

End Sub
```

Branje barve za nas opravi kar vgrajena grafična metoda Point. Iz zgleda je razvidno, da lahko grafično površino gradnika uporabljamo povsem enako kakor grafično površino obrazca.

Če bomo nov gradnik preizkusili v testnem projektu, bomo v oknu s kodo med dogodki gradnika našli tudi pravkar pripravljeni dogodek ColorSelected. To potrjuje tezo, da so vsi gradniki samodejno, a nevidno napovedani v obrazcu z določilom WithEvents, kar olajša njihovo rabo.



Slika 18-33: Nov dogodek lastnega gradnika

NAMIG Programsko kodo zgleda najdete na priloženi plošči CD-ROM v mapi \Primeri\4-Razredi\2-Gradniki\3-ActiveX Control\2-Izdelava\2-Preliv\.

Napotki

Izdelovanje lastnih gradnikov je ena najbolj vznemirljivih možnosti visual basica. Hkrati je tudi ena najzahtevnejših, saj zahteva temeljito poznavanje načrtovanja obrazcev in predmetnega programiranja. Zgleda, ki smo ju že spoznali, ponujata dovolj informacij za začetek razvoja novih gradnikov. Tu si bomo ogledali še nekaj splošnih napotkov, ki nam pomagajo pri izgradnji najrazličnejših vrst gradnikov.

Dogodki, dogodki

Pri spoznavanju z lastnimi gradniki smo večkrat omenili pomen posebnih dogodkov, ki se jim lahko pripetijo in se moramo nanje ustrezno odzvati. Tu jih še enkrat pregledno naštejmo:

Dogodek	Opis
Initialize	Dogodek se pripeti ob *vsaki* vzpostavitvi gradnika. Vedno je prvi dogodek v njegovem življenjskem krogu.
InitProperties	Dogodek se pripeti samo ob *prvi* vzpostavitvi posameznega izvoda gradnika. Namenjen je vzpostavitvi lastnosti na njihove privzete vrednosti.
ReadProperties	Dogodek se pripeti vedno, kadar je treba uskladiti podobo gradnika s prej shranjenimi vrednostmi lastnosti. Odziv nanj uporabimo za branje shranjenih lastnosti.
Resize	Dogodek se pripeti po vsaki spremembi velikosti gradnika. Odziv na ta dogodek je posebej pomemben pri gradnikih, ki združujejo enega ali več obstoječih, temeljnih gradnikov. V odzivni proceduri uskladimo njihove velikosti in položaj z novo velikostjo gradnika.
Paint	Najpomembnejši dogodek pri lastnih gradnikih, katerih vizualno podobo ustvarjamo sami. Dogodek se lahko pripeti kadarkoli, njegova odzivna procedura pa mora poskrbeti za izris gradnika v trenutnem stanju.
WriteProperties	Kadarkoli se vrednost kake lastnosti spremeni, visual basic to zazna in ob uničenju izvoda gradnika pokliče odzivno proceduro dogodka WriteProperties. V njej shranimo nastavitve vseh lastnosti. Ker pri shranjevanju lahko podamo tudi privzeto vrednost, bo razvojno okolje samo izločilo vrednosti, ki so se dejansko spremenile.
Terminate	Vedno zadnji dogodek, ki pospremi uničenje posameznega izvoda gradnika.

Poleg zgoraj naštetih je včasih treba paziti tudi na dogodka Show in Hide.

Dogodek	Opis
Show/Hide	Dogodka se sprožita, kadar se spremeni vidnost gradnika, torej njegova lastnost Visible. Dogodkov *ne* bomo prejeli ob zakritju ali minimiziranju vsebnika.

Razvoj gradnika je v marsičem podoben razvoju obrazca, vendar je prav pri dogodkih nekaj pomembnih razlik. Seveda gradnik pozna posebne dogodke za delo z lastnostmi, omenjene tudi v zgornji tabeli. Po drugi strani pa ne pozna dogodkov Activate in Deactivate, ker gradniki prejemajo vnosno točko in ne postanejo dejavni tako kakor obrazci.

Vnosna točka

Prejem in izgubo vnosne točke (*focus*) ponazarjata dogodka GotFocus in LostFocus. V lastnem gradniku, ki ga izrisujemo sami, moramo ta dva dogodka izkoristiti za morebitno spremembo vizualne podobe gradnika (npr. izris črtkastega okvira, na pomoč lahko pokličemo funkcijo API DrawFocusRect), nikoli pa ju ne smemo prožiti naprej. Za prenos dogodka ob spremembi vnosne točke poskrbi že sam vsebnik. Če ne želimo, da naš gradnik prejme vnosno točko, v oknu za nastavljanje lastnosti izključimo lastnost **CanGetFocus**. Tovrstne gradnike lahko uporabimo za posredovanje vnosne točke naslednjemu gradniku v dostopnem zaporedju. V tem primeru nastavimo lastnost **ForwardFocus**. Gradnik se bo sedaj pri vnosni točki vedel podobno oznaki (*Label*).

Če je gradnik sestavljen iz enega ali več obstoječih, temeljnih gradnikov, ne bo mogel prejeti vnosne točke, ne glede na nastavitev lastnosti CanGetFocus. Sam gradnik lahko prejme vnosno točko le, če ga sestavljajo samo taki temeljni gradniki, ki sami niso sposobni prejeti vnosne točke (npr. lik). Ko je na lastnem gradniku nameščen vsaj en temeljni gradnik, ki lahko prejme vnosno točko, mora biti lastnost CanGetFocus postavljena na resnično vrednost. Kot smo že omenili, gradnik v tem primeru ne more prejeti vnosne točke, zato se tudi ne prožita njegova dogodka GotFocus in LostFocus, zato pa prejme dve drugi obvestili:

- **EnterFocus**. Dogodek se pripeti, ko prvi izmed temeljnih gradnikov prejme vnosno točko. Ker se pripeti, preden prvi temeljni gradnik prejme dogodek GotFocus, lahko odziv nanj uporabimo za preusmeritev vnosne točke na poljubni izbrani temeljni gradnik.
- **ExitFocus**. Dogodek pospremi izgubo vnosne točke pri zadnjem temeljnem gradniku. Odziv nanj lahko uporabimo za shranjevanje gradnika, ki je nazadnje imel vnosno točko.

Seveda vsi temeljni gradniki prejemajo dogodke GotFocus in LostFocus, ki jih v programski kodi lastnega gradnika lahko prestrežemo in uporabimo.

Lastnim gradnikom lahko omogočimo tudi dostopno tipko, kot jo poznajo menuji, ukazni gumb, stikala in oznake. Dostopna tipka je določena z lastnostjo **AccessKeys**, ki ji lahko priredimo celo niz znakov. Kadarkoli uporabnik pritisne tipko Alt skupaj z eno izmed tipk, shranjenih v omenjeni lastnosti, gradnik prejme dogodek **AccessKeyPress** skupaj s parametrom, ki ponazarja pritisnjeno tipko. Preostane nam samo, da s kodo izluščimo ustrezne dostopne tipke (npr. z analizo niza v neki javni lastnosti, npr. Caption, in položaja znaka & v njem), jih priredimo lastnosti AccessKeys ter v odzivu dogodka AccessKeyPress po potrebi preusmerimo vnosno točko na izbrani temeljni gradnik.

Poravnava

Nekateri gradniki omogočajo poravnavo ob rob svojega vsebnika z lastnostjo Align. Če jo želimo podpreti tudi z lastnim gradnikom, preprosto vklopimo lastnost **Alignable**. Če različne poravnave vplivajo na videz gradnika, lahko z lastnostjo Align vgrajenega predmeta Extender preverjamo morebitno poravnavo.

> **NAMIG** Še enkrat: lastnosti predmeta Extender je treba uporabljati previdno (beri: z vključenim prestrezovalnikom izvajalnih napak).

Prosojnost

Izdelovanje lastnih gradnikov je še privlačnejše zaradi možnosti, da ozadje gradnika napravimo prosojno in tako izdelamo gradnike najrazličnejših oblik. Prosojnost dosežemo zelo preprosto, gradniku nastavimo lastnost **Transparent** na resnično vrednost. V tem primeru postane površina gradnika, ki je ne prekriva kateri izmed temeljnih gradnikov, prosojna. Dogodki nad to površino, povzročeni z miško, se posredujejo vsebniku in jih gradnik ne zazna.

Če je kateri izmed vsebovanih gradnikov tudi delno prosojen, se njihova prosojnost prav tako upošteva. Prosojen je lahko gradnik lik (**Shape**) z ustrezno nastavljeno lastnostjo BackStyle in celo oznaka (Label), vendar le, če za izpis uporablja pisavo TrueType. V tem primeru se upoštevajo celo luknjice v črkah, kot je B. To pa še ni vse, samemu gradniku lahko v lastnosti MaskBitmap priredimo bitno sliko, ki prekriva želeno prosojno območje s poljubno barvo. To barvo priredimo lastnosti MaskColor[66].

Ugotavljanje prosojnega območja lahko od izvajalnega okolja terja precej računanja, zato ga uporabljajmo previdno.

[66] Če uporabimo bitno sliko v zapisu, ki sam po sebi podpira prosojnost, npr. GIF ali ICO, lastnosti MaskColor ni treba nastavljati.

O(ne)mogočanje

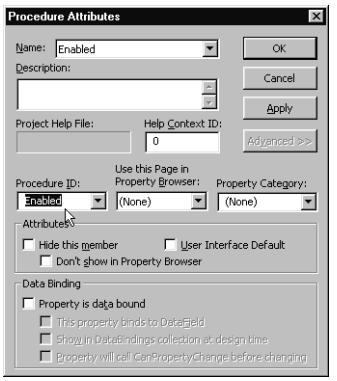
Omogočenost gradnika navadno določa lastnost Enabled. Pri lastnih gradnikih lahko omogočenost ugotavljamo z istoimensko lastnostjo predmeta Extender, ki pa je na voljo samo tedaj, ko tudi gradnik izpostavlja enako poimenovano lastnost. In ne samo to, podprograma lastnosti Enabled je treba označiti s posebno vrednostjo (procedure ID)! Ustrezen postopek za objavo lastnosti Enabled se glasi:

Najprej zapišemo ustrezne podprograme lastnosti, ki delo preložijo na lastnost Enabled predmeta UserControl:

```
Public Property Get Enabled() As Boolean
  Enabled = UserControl.Enabled
End Property

Public Property Get Enabled(ByVal NewEnabled As Boolean)
  UserControl.Enabled = NewEnabled
  PropertyChanged "Enabled"
End Property
```

S spreminjanjem omogočenosti predmeta UserControl o(ne)mogočamo vse vsebovane temeljne gradnike. Dela še nismo končali, podprograma lastnosti Enabled moramo posebej označiti. V glavnem menuju Tools izberemo ukaz **Procedure Attributes**... V prikazanem pogovornem oknu z zaprtega seznama **Name** izberemo lastnost »Enabled« in kliknemo gumb **Advanced**. Prikaže se preostali del pogovornega okna, med drugim tudi zaprti seznam **Procedure ID**, v katerem izberemo oznako **Enabled**.

Slika 18-34: Določanje oznake ustrezni lastnosti



S prirejanjem te oznake zasenčimo lastno lastnost Enabled z vsebnikovo. S tem bo naš gradnik postal o(ne)mogočen skupaj s svojim vsebnikom. Vendar pa tudi onemogočeni gradnik ni povsem onemogočen: ne prejema sicer dogodkov[67], vendar se mora ustrezno upodobiti (ukazni gumb npr. posivi svoj napis). Pri gradnikih, ki

[67] Ker jih ne prejema sam vsebnik.

jih izrisujemo z lastno kodo, moramo zato preverjati nastavitev lastnosti UserControl.Enabled:

```
Private Sub UserControl_Paint()
  If UserControl.Enabled
    ' izriši omogočeno podobo
  Else
    ' izriši onemogočeno podobo
  End If
End Sub
```

Nevidnost med izvajanjem

Nekateri gradniki so vidni le med načrtovanjem vmesnika, saj ponujajo posebne storitve, ki ne zahtevajo uporabniškega vmesnika. Tovrstne gradnike lahko razvijemo tudi sami. Preden pa se lotimo razvoja tovrstnega gradnika, razmislimo, ali bi ne bilo bolje uporabiti kar znotrajprocesnega strežnika ActiveX.

Med izvajanjem nevidni gradnik označimo z nastavitvijo lastnosti **InvisibleAtRuntime** na resnično vrednost. Za *trajno* nevidnost ni priporočljivo uporabljati lastnosti Visible predmeta Extender. Nevidni gradniki med načrtovanjem praviloma niso povečljivi, kar dosežemo z naslednjo odzivno proceduro:

```
Private Sub UserControl_Resize()
  Size 420, 420
End Sub
```

> **NAMIG** Ne pozabimo: skala koordinatnega sistema lastnega gradnika so vedno *twipi*.

Vezani gradnik

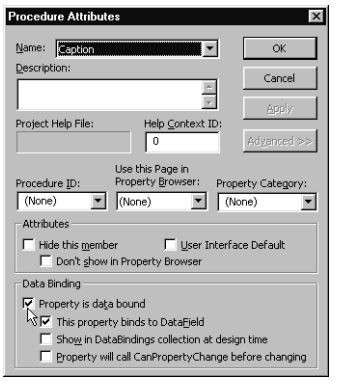
Določene lastnosti gradnika lahko označimo kot *povezljive*. Povezljivo (*bindable*) lastnost lahko uporabnik gradnika poveže s poljem v zbirki podatkov. Tovrstne lastnosti označimo z ukazom **Procedure Attributes**... iz glavnega menuja **Tools**.

Želeno lastnost izberemo s seznama **Name** in prikažemo preostanek pogovornega okna s klikom gumba **Advanced**. Z vklopom stikala **Property Is Data Bound** označimo, da je lastnost povezljiva. S tem omogočimo še druga stikala v skupini. Njihov pomen je zbran v spodnji tabeli:

Stikalo	Pomen (če je vključeno)
This property binds to DataField	Izbrana lastnost bo povezana s poljem, ki ga določa lastnost DataField. Tako se vede večina že pripravljenih vezanih gradnikov. To stikalo lahko

Stikalo	Pomen (če je vključeno)
	vključimo za eno samo lastnost. Ponavadi izberemo tisto, ki najočitneje izraža povezavo z zbirko podatkov.
Show in DataBindings collection at design time	Vse povezane lastnosti so vključene v zbirko DataBindings, ki je dostopna iz programske kode. Če želimo povezano lastnost prikazati tudi med načrtovanjem, vklopimo to stikalo. Povezava tako označenih lastnosti bo mogoča s posebnim urejevalnikom lastnosti DataBindings.
Property will call CanPropertyChange before changing	Lastnosti, ki so izpostavljene navzven in jih lahko spremeni tudi njihov uporabnik, bi morale vprašati metodo CanPropertyChange, katere odgovor določa, ali je določeno lastnost mogoče popraviti. Če se v programski kodi zares držimo tega pravila, vključimo to stikalo in s tem omogočimo bolj trdoživo delovanje.

Slika 18-35: Določanje povezljive lastnosti



Oglejmo si zgled lastnosti Text, ki je izpostavljena uporabniku in hkrati vezana na zbirko podatkov:

```
Public Property Let Text(NewText As String)
  If CanPropertyChange("Text") Then
    txtVnos.Text = NewText
    ' moramo poklicati metodo PropertyChanged
    ' sicer se podatkovni vir ne bo posodobil
    PropertyChanged("Text")
  End If
End Property
```

V odzivni proceduri za vsa opravila zadolžimo vnosno polje. Nujno je, da pokličemo metodo PropertyChanged. Tako bo visual basic poskrbel, da se v ustreznem trenutku osveži tudi vsebina v podatkovnem viru (npr. ob premiku na nov zapis v zbirki zapisov). Ker lahko uporabnik besedilo spremeni tudi neposredno, moramo omenjeno metodo poklicati tudi v odzivu na dogodek Change samega vnosnega polja:

```
Private Sub txtVnos_Change()
  ' moramo poklicati metodo PropertyChanged
  ' sicer se podatkovni vir ne bo posodobil
  PropertyChanged("Text")
End Sub
```

Dodatna oprema

Lasten gradnik lahko opremimo tudi s posebno bitno sliko, ki ga bo ponazarjala v orodjarni, in z morebitno vizitko, obrazcem, ki izpiše osnovne podatke o samem gradniku. Bitna slika v orodjarni je vedno velikosti 16 × 15 pik, zato je ne pripravljamo v zapisu ICO (*Icon*), ampak raje uporabimo obliko BMP (*Bitmap*), najbolje v 16 osnovnih barvah, ki jih podpira vsak sistem v Oknih. Pripravljeno bitno sliko priredimo lastnosti **ToolboxBitmap**.

Vizitko oblikujemo na posebnem obrazcu, ki ga vključimo v obstoječi projekt vrste »ActiveX Control«. Če želimo, lahko izdelamo posebne vizitke za vsak gradnik, shranjen v projektu, ali pa uporabimo eno samo, generično vizitko. Ko smo končali načrtovanje obrazca, gradniku dodamo novo *javno* proceduro, ki ji priredimo ustrezno oznako. Procedura bi se lahko glasila takole:

```
Public Sub AboutBox()
  frmAbout.Show vbModal
  Unload frmAbout
  Set frmAbout = Nothing
End Sub
```

Koristno je, da se vizitke po prikazu popolnoma znebimo, da po nepotrebnem ne zaseda prostora v pomnilniku. Z ukazom **Procedure Attributes**... iz glavnega menuja **Tools** sedaj tej javni proceduri dodelimo oznako **AboutBox** na seznamu **Procedure ID**. Gradnik bo imel sedaj v oknu za nastavljanje lastnosti posebno lastnost, simbolično označeno z »(About)«. Urejanje te lastnosti pravzaprav prikliče pripravljeno vizitko.

Prevajanje

V posameznem projektu vrste »ActiveX Control« lahko določimo enega ali več lastnih gradnikov ter poljubno število strani lastnosti, obrazcev in samostojnih modulov s kodo. Vse skupaj lahko prevedemo v samostojno obliko, datoteko s podaljškom OCX. Projekt z gradniki prevedemo povsem enako kakor katerikoli drug projekt. Najprej ga izberemo v projektnem oknu, zatem pa v menuju **File** izberemo ukaz **Make Project1.ocx**... (ime ukaza je odvisno od imena projekta).

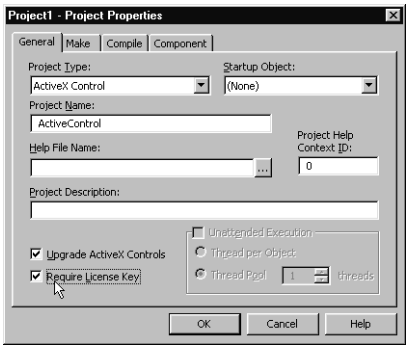
Pred prevajanjem vsakega gradnika ActiveX lahko nastavimo nekaj lastnosti projekta, ki vplivajo na poznejšo uporabo gradnikov. Spoznali jih bomo v naslednjem poglavju.

Dovoljenja

Visual basic omogoča zaščito uporabe razvitega gradnika s posebnim dovoljenjem (*license key*), ki se zapiše v zunanjo datoteko in posebej označi v prevedeni datoteki gradnika. Datoteko z dovoljenjem zna orodje Application Setup Wizard vključiti v namestitveni postopek. Ob namestitvi gradnika v ciljni računalnik se to dovoljenje prepiše v tamkajšnji sistemski register in dopusti uporabo gradnika v razvojnih orodjih. Če nekdo zaščiteni prevedeni gradnik v datoteki OCX zgolj prepiše v drug računalnik, ga v razvojnem okolju ne bo mogel uporabiti.

Zapis dovoljenja zahtevamo z vklopom stikala **Require License Key**, ki ga najdemo pod jezičkom **General** v pogovornem oknu z nastavitvami lastnosti projekta (**Project properties**).

Slika 18-36: Uporaba gradnika v razvojnih okoljih bo zahtevala posebno dovoljenje, zapisano v registru.



Dovoljenje za uporabo vpliva zgolj na rabo gradnika v razvojnih orodjih. Gradnik brez dovoljenja še vedno lahko uporabljajo vsi že izdelani programi, ki zahtevajo njegove storitve, onemogočeno pa je njegovo nadgrajevanje in uporaba v novih projektih.

19. POGLAVJE

Razpečevanje

Priprava namestitvene različice za gradnike ActiveX je še posebej zapleteno opravilo, saj je treba upoštevati kopico napotkov v najrazličnejših datotekah projekta. Na srečo je vsemu temu delu uspešno kos priloženi pripomoček Application Setup Wizard. V tem poglavju si bomo ogledali značilnosti njegove uporabe pri razpečevanju gradnikov in tudi omenili posebnosti pri pripravi namestitvene različice *odjemalca*, ki uporablja storitve katerega izmed strežnikov ActiveX.

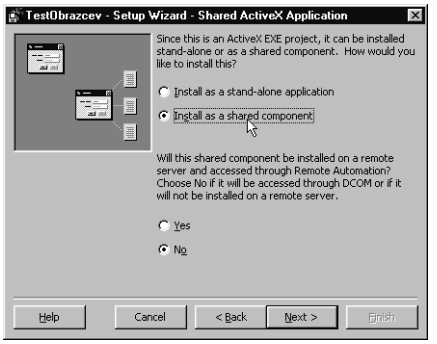
Vklop strežnika

Če naš program uporablja storitve posebnega strežnika ActiveX, prav tako razvitega v visual basicu, nam pripomoček Application Setup Wizard omogoča izdelovanje namestitvene procedure, ki bo poskrbela tudi za ustrezno namestitev strežnika.

Namestitev zunajprocesnega strežnika

Za uspešno uporabo zunajprocesnega strežnika (»ActiveX EXE«) v drugih računalnikih moramo najprej pripraviti njegovo namestitveno proceduro. Pripravimo jo s pripomočkom Application Setup Wizard. Postopek je v začetku podoben izdelovanju običajne namestitvene procedure. V prvem koraku vnesemo ime projektne datoteke z opisom zunajprocesnega strežnika in preverimo, ali je vklopljeno stikalo **Create a Setup Program**. Naslednji štirje koraki so enaki kakor pri običajnih projektih, zatem pa sledi posebno pogovorno okno, v katerem določimo, ali se naj strežnik namesti kot samostojni program (*stand-alone application*) ali kot deljeni gradnik (*shared component*), ki zgolj skrbi za oskrbovanje odjemalcev. V tem koraku lahko tudi določimo, ali se bo strežnik nameščal krajevno (stikalo **No**) ali pa bo nameščen v drugem računalniku in bo svoje storitve izpostavljal prek oddaljene avtomatizacije (stikalo **Yes**). Videz pogovornega okna je razviden s slike 19-1.

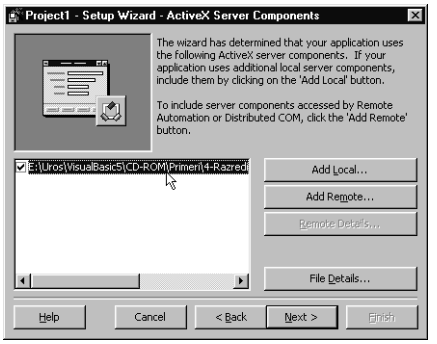
Preostanek priprave namestitvenega postopka se ne razlikuje od priprave namestitve običajnih projektov.



Slika 19-1: Posebna določila za nameščanje zunajprocesnega strežnika

Namestitev odjemalca

Izdelava namestitvenega postopka, ki uporablja storitve strežnikov ActiveX, prav tako zahteva nekaj dodatnih korakov. Vsak odjemalec storitev mora biti vsaj enkrat preveden v izvršljivo datoteko že v razvojnem okolju visual basica. Če odjemalec uporablja storitve kakega *oddaljenega* strežnika, moramo že prej pripraviti ustrezne datoteke VBR (glej obravnavo napotkov za strežnike v prejšnjem poglavju, pod naslovom Strežniki v strežniku).



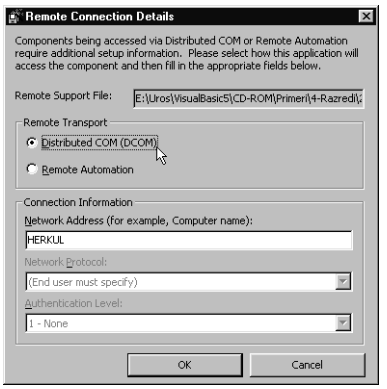
Slika 19-2: Dodajanje uporabljenih strežnikov v namestitveni postopek

Izdelovanje namestitvenega postopka je v začetku enako kakor pri običajnih projektih. Po prvih treh korakih pa se prikaže pogovorno okno s seznamom zaznanih *krajevnih* strežnikov, ki jih naš projekt uporablja. Če na seznamu manjka kak strežnik (zagotovo se to zgodi pri uporabi morebitnih oddaljenih strežnikov), ga lahko dodamo.

- Krajevne strežnike dodamo z gumbom **Add Local**, ki prikliče standardno pogovorno okno za odpiranje datoteke EXE oziroma DLL.
- Oddaljene strežnike dodamo z gumbom **Add Remote**. Ta gumb prikliče standardno pogovorno okno, s katerim najdemo ustrezno datoteko VBR, potrebno za izdelavo pravilnega namestitvenega postopka.

Če v namestitveni postopek dodamo kak oddaljeni strežnik, se nemudoma prikaže pogovorno okno, ki zahteva dodatne informacije, potrebne za uspešno povezavo z njim. Vidno je na naslednji sliki:

Slika 19-3: Podrobnosti povezave z oddaljenim strežnikom



Če za navezavo uporabimo posredovanje DCOM, zadošča, da v vnosno polje **Network Address** vpišemo še opisno ime strežnega računalnika. Pri uporabi oddaljene avtomatizacije (*Remote Automation*) določimo tudi omrežni protokol in način prijave.

Po vnašanju zahtevanih podatkov sta na seznamu lahko tako krajevna različica strežnika (datoteka EXE) kakor opis priklopa nanj kot oddaljeni strežnik (datoteka VBR). V tem primeru ročno izključimo krajevno različico (datoteko EXE).

Preostanek priprave namestitvenega postopka se ne razlikuje od priprave namestitve običajnih projektov.

Prenos prek spleta

Gradnike ActiveX je mogoče uporabiti tudi znotraj spletnih strani, prikazanih v brskalniku Microsoft Internet Explorer 3.0 ali poznejšem. Za njihovo rabo moramo pripraviti namestitev v posebni obliki, ki podpira prenos najnujnejših datotek v stisnjeni obliki posebne datoteke vrste CAB kar prek spleta (*word wide web*).

Gradnik, katerega namestitev pripravimo v tej obliki, se v eni sami datoteki prenese v ciljni računalnik, razširi, preveri, prijavi v registru, namesti in zatem še požene.

NAMIG Uporaba gradnikov ActiveX je znotraj spletnih brskalnikov zaenkrat praktično uporabna le v hitrih intranetih na »platformi« Wintel32. Pesti jih predvsem podpora samo eni platformi, slaba varnost in prevelika prostorska zahtevnost (velikost).

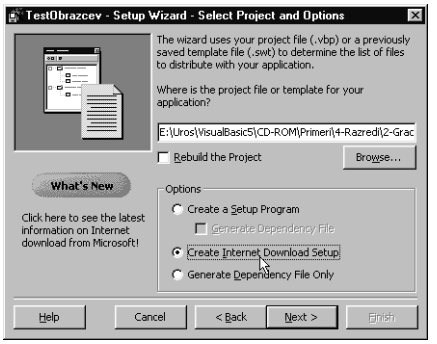
Pripomoček za izdelovanje namestitvenih postopkov, Application Setup Wizard, zna pripraviti vse potrebno za tovrstno razpečevanje izdelanih gradnikov. Pripravi celo vzorčno stran v označevalnem jeziku HTML, ki rabi za preizkus gradnika.

V prvem koraku izdelave namestitvenega postopka vklopimo stikalo **Create Internet Download Setup**. S tem smo izbrali izdelavo *primarne* datoteke CAB z naslednjimi elementi:

- Elementi projekta, torej izbranega gradnika ActiveX.
- Datoteka INF, ki vsebuje povezavo z drugimi potrebnimi (*sekundarne*) datotekami CAB in nekaterimi dodatnimi informacijami o projektu. Ta datoteka nadomesti nadzorno datoteko Setup.LST, ki nadzira nameščanje v običajnih postopkih.
- Rezerviran prostor za digitalne podpise.
- Vse preostale datoteke, ki še niso vključene v kako drugo (*sekundarno*) datoteko CAB.

Naslednji korak spet vpraša po ciljni mapi, v katero se bodo shranile datoteke izdelanega namestitvenega postopka.

Slika 19-4: Prvi korak pri izdelovanju namestitve za splet



V tretjem koraku se odločimo, od kod bomo črpali sekundarne datoteke CAB in določimo dodatne atribute projekta. Pogosto rabljene datoteke za gradnike ActiveX, kot so nujno potrebna knjižnica Msvbvm50.DLL, zunanji gradniki, razredna knjižnica DAO itn., so pripravljene v posebnih, sekundarnih datotekah CAB, ki so digitalno podpisane in shranjene na spletni strani podjetja Microsoft. V datoteki INF, ki nadzira postopek nameščanja, so lahko povezave s temi datotekami, ali pa se odločimo, da jih bomo že prej prenesli iz interneta v krajevne izvode v našem omrežju. Ta možnost je še posebej privlačna za intranete, ki jih ne želimo odpirati v internet.

Slika 19-5: Določanje nahajališča sekundarnih datotek CAB

V tretjem koraku priprave namestitvenega postopka za splet lahko izberemo stikalo **Dowload from Microsoft web site**, če želimo potrebne datoteke prenesti neposredno z Microsoftove spletne strani, ali pa vklopimo stikalo **Use alternate location** in v vnosno polje vnesemo naslov strežnika s kopijo teh datotek. Naslov je lahko podan v obliki URL ali UNC.

Po želji lahko določimo še varnostne atribute našega gradnika. S klikom gumba **Safety** prikličemo posebno pogovorno okno, v katerem določimo, ali je naš gradnik varen za uporabo s kakršnimikoli vhodnimi podatki (stikalo **Safe for initialization**) in ali je varen za uporabo s katerimkoli skriptnim jezikom, ki ga lahko nadzira (stikalo **Safe for scripting**). Če bomo potrdili varnost in pripravljeni paket digitalno podpisali, se bo uporabnik gradnika lahko odločil, ali verjame naši izjavi.

> **NAMIG** Seveda imajo različni digitalni podpisi pri uporabnikih (in pri nekateri programski opremi, ki samodejno preverja podpise) različno težo. Microsoft Internet Explorer seveda skoraj slepo verjame Microsoftovemu digitalnemu podpisu.

Preostanek priprave namestitvenega postopka se ne razlikuje od priprave namestitve običajnih projektov.

Na koncu nas v ciljni mapi pričakajo vse potrebne datoteke, vključno z vzorčno stranjo v označevalnem jeziku HTML, ki jo lahko uporabimo kot smerokaz do pravkar pripravljenega namestitvenega postopka. V ciljni mapi je tudi mapa Support, v kateri so vse datoteke, stisnjene v datoteko CAB, v izvorni, nestisnjeni obliki. Datoteke v tej mapi lahko posodabljamo (predvsem to velja za nadzorno datoteko INF) in ročno stisnemo v datoteko CAB s pripomočkom Makecab, ki ga najdemo v mapi Setupkit\Kitfil32\ znotraj mape, v kateri je nameščen sam visual basic.

PETI DEL

Svet

V TEM DELU

Izpostavljeni razredi
Narečja

Razvojno okolje visual basic še zdaleč ni edini kraj, kjer lahko pišemo kodo v tem posebnem narečju jezika basic in uporabljamo strežnike in gradnike ActiveX. Osnovni del visual basica, razredna knjižnica Visual Basic for Application (VBA) in posebno razvojno okolje, je vključen v skupek namenskih programov Microsoft Office, podjetje Microsoft pa je tehnologijo VBA ponudilo v odkup tudi vsem zainteresiranim razvijalcem programske opreme za Okna. Med drugim imata dovoljenje za njegovo uporabo že zelo znani podjetji Autodesk in Visio.

A tu se pohod basica še ne konča. Še bolj okrnjena različica, Visual Basic Scripting (VBS), si je utrla pot v nadzorovanje sistemskih orodij. S posebnim gostiteljem, Windows Scripting Host, ki je na voljo kot dodatek za obstoječe operacijske sisteme Okna, v prvi prenovi pa bo postal njihov sestavni del, uporablja VBS za sistemsko upravljanje, spletni strežnik Internet Information Server pa znotraj dejavnih spletnih strani, ki omogočajo pripravo dinamičnih spletnih predstavitev.

NAMIG Spletni strežnik in gostitelj Windows Scripting Host najdete tudi na priloženi plošči CD-ROM. Več o njuni rabi si preberite v drugem dodatku.

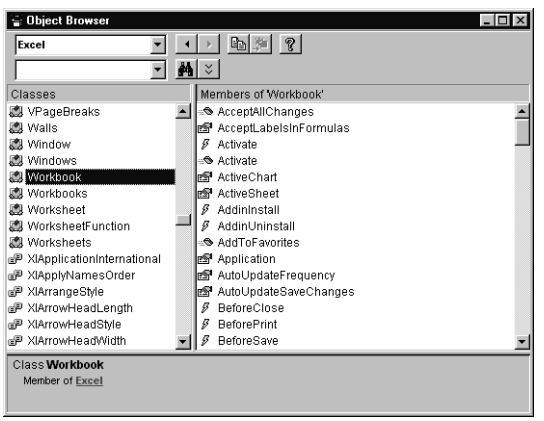
Hkrati je visual basic odprt tudi v drugo smer. Že večkrat smo omenili, da je večina sodobnih namenskih programov za operacijski sistem Okna izdelanih v obliki samostojnih zunajprocesnih strežnikov vrste »ActiveX EXE«. Zato lahko njihove izpostavljene razrede uporabimo v vsakem razvojnem okolju, ki podpira programirano avtomatizacijo. Če imamo v našem računalniku ali v krajevnem omrežju nameščene tovrstne programe, jih lahko izkoriščamo kar v visual basicu. Zakaj bi npr. razvijali pregledovalnik pravopisa, če lahko uporabimo kar izpostavljeni razred Microsoftovega Worda?

20. POGLAVJE

Izpostavljeni razredi

Če imamo v našem računalniku ali v strežniškem računalniku v omrežju, v katerega smo vključeni, nameščeno programsko opremo, ki ustrezno izpostavlja svoje razredne knjižnice, lahko njihovo funkcionalnost v celoti izkoristimo iz visual basica. Prav z rabo izpostavljenih knjižnic lahko zaslutimo moč modela gradnikov, saj nam praktično ni treba več programirati nove funkcionalnosti, ampak samo združujemo že pripravljene kose programske kode. Seveda ima uporaba zunanjih razredov tudi slabo stran. Program, ki jih uporablja, zahteva, da je enaka razredna knjižnica na voljo tudi v vseh ciljnih računalnikih, v katere jo želimo namestiti. To pa lahko pomeni precejšnjo naložbo v nakup dovoljenj za uporabo ustreznega števila izvodov namenskih programov.

Kot vemo, lahko izpostavljene razredne knjižnice, prijavljene v sistemskem registru, najdemo v seznamu pogovornega okna ukaza **Project ➪ References...** Preden uporabimo posamezen razred, moramo njegovo knjižnico najprej vključiti v projekt. Po vključitvi lahko razrede in njihove vmesnike pregledujemo s predmetnim brskalnikom (*object browser*). Žal nam ta brskalnik ne more ponuditi vseh potrebnih informacij o uporabi gradnika, zato moramo pred resno uporabo dobiti še ustrezno dokumentacijo, bodisi pisno bodisi elektronsko.



Slika 20-1: Izpostavljena razredna knjižnica programa Microsoft Excel 8.0 (97), prikazana v predmetnem brskalniku.

Zaradi združljivosti s starejšim modelom avtomatizacije (OLE 2) je mogoče uporabljati tudi razrede, za katere ni na voljo posebne razredne knjižnice. Njihovo rabo bomo natančneje spoznali v nadaljevanju.

Uporaba gradnikov

Uporaba izpostavljenih razredov, ki jih priskrbijo gradniki ActiveX, se ne razlikuje dosti od uporabe kateregakoli drugega razreda v visual basicu. Najprej moramo pravilno vzpostaviti ustrezno predmetno spremenljivko, zatem lahko z njo uporabljamo vmesnik razreda, na koncu pa moramo spremenljivko ustrezno sprostiti in s tem posredno sprostiti tudi strežnik. Pravzaprav uporaba izpostavljenih razredov za nas ni nič posebej novega, v tretjem delu knjige smo že uporabljali predmetni model DAO, ki je pravzaprav posebna razredna knjižnica.

Za hitro in učinkovito demonstracijo si oglejmo uporabo razredne knjižnice preglednice Microsoft Excel, ki je v bistvu samostojen, zunajprocesni strežnik ActiveX. Naslednji zgled bo prebral vrednosti iz uporabniškega vmesnika projekta v visual basicu, jih posredoval preglednici Excela, ki ju bo seštela, ter prebral in izpisal izid:

```
Private Sub Command1_Click()
' Napoved spremenljivk
Dim xlApp As Excel.Application
Dim xlBook As Excel.Workbook
Dim xlSheet As Excel.Worksheet
Dim izid

  ' Stvaritev potrebnih predmetov
  Set xlApp = Excel.Application
  Set xlBook = xlApp.Workbooks.Add
  Set xlSheet = xlBook.Worksheets.Add

  ' Uporaba predmeta - preglednice
  With xlSheet
    .Cells(1, 1).Value = Text1.Text
    .Cells(2, 1).Value = Text2.Text
    .Cells(3, 1).Formula = "=R1C1+R2C1"
    izid = .Cells(3, 1)
    .SaveAs "c:\Zvezek.xls"
  End With

  ' Sprostitev aplikacije
  xlApp.Quit

  ' Sprostitev referenc
  Set xlSheet = Nothing
  Set xlBook = Nothing
  Set xlApp = Nothing
```

```
  ' Izpis izida
  MsgBox izid

End Sub
```

Če zapisani zgled preizkusimo, se v ozadju nevidno požene sicer samostojni strežnik ActiveX (seveda, če Excel že ni pognan od prej), ki opravi naloženo opravilo in se na koncu pospravi iz pomnilnika. Ko bi ne bilo krajše zamude pri nalaganju strežnika, bi bila njegova uporaba navzven praktično neopazna.

> **NAMIG** Zaradi preprostosti zgornji zgled ne uporablja prestrezovalnika izvajalnih napak, čeprav je ta pri uporabi izpostavljenih razredov skoraj nujen. Njegovo uporabo bomo spoznali v nadaljevanju.
>
> Zglede iz tega poglavja najdete na priloženi plošči CD-ROM v mapi \Primeri\5-Svet\1-Izpostavljeni razredi\.

Sklicevanje

Pred uporabo lastnosti in metod predmeta moramo vzpostaviti ustrezno predmetno spremenljivko. Njena vzpostavitev je odvisna od dveh dejavnikov:

- Ali je za določen predmet na voljo razredna knjižnica ali ne.
- Ali je določen predmet mogoče ustvariti neposredno ali gre za posredno ustvarljiv predmet, ki je pravzaprav lastnost katerega od neposredno ustvarljivih predmetov.

V zadnjem zgledu opazimo, da je predmet razreda Excel.Application mogoče ustvariti neposredno (*externally creatable object*), druge predmete pa smo ustvarjali sklicujoč se na njegove lastnosti in metode.

Če za razred ni na voljo ustrezna predmetna knjižnica, moramo uporabiti predmetno spremenljivko generičnega tipa Object, kar pomeni pozno povezovanje in zato manjšo učinkovitost programske kode. No, za izkoriščanje razredov starejših strežnikov ActiveX (takrat so se imenovali še strežniki OLE) je to edina pot.

> **NAMIG** Zaradi razreševanja imenskih konfliktov je najbolje, da vse razrede iz knjižnic v napovedih opišemo z njihovim popolnim imenom, torej v obliki `knjižnica.razred`.

Predmetno spremenljivko je mogoče vzpostaviti na več načinov:

- Z implicitno napovedjo z določilom New. Ta pristop ni priporočljiv, saj je manj učinkovit in lahko zaplete sproščanje vseh referenc, kar je nujni pogoj za sprostitev samega strežnika.

- Z uporabo stavka Set, v katerem lahko uporabimo operator New ali funkcijo **CreateObject**. S tem lahko vzpostavimo spremenljivko, ki se sklicuje na nov izvod razreda.
- Z uporabo stavka Set in funkcije **GetObject**. Tako lahko ustvarimo spremenljivko, ki se sklicuje na nov ali že obstoječi izvod razreda.

Uporaba določila/operatorja New je mogoča le za razrede, ki pripadajo razredni knjižnici, vključeni v projekt. Drugače je pri funkciji CreateObject. Uporabimo jo lahko za stvaritev vsakega avtomatiziranega predmeta, ne glede na to, ali je njegova razredna knjižnica vključena v projekt. Funkcija kot edini parameter prejme niz znakov, ki mora vsebovati popolno programsko oznako razreda, v obliki `strežnik.oznakaRazreda`. Žal oznaka razreda ni vedno enaka imenu razreda, kot se pojavlja v razredni knjižnici (no, v teh primerih lahko tako ali tako posežemo po operatorju New). Za zgled ustvarimo razred, ki predstavlja celotni program Microsoft Excel:

```
Dim xlApp As Object
  Set xlApp = CreateObject("excel.application")
```

Če bi v projekt vključili še ustrezno razredno knjižnico, bi lahko zapisali učinkovitejšo obliko kode:

```
Dim xlApp As Excel.Application
  Set xlApp = CreateObject("excel.application")
```

Oglejmo si še uporabo funkcije GetObject. To praviloma uporabljamo za vzpostavitev novega sklicevanja na že izdelani predmet, čeprav lahko z njo ustvarjamo tudi čisto nove izvode. Njena popolna oblika se v simboličnem zapisu glasi:

```
Set predmet = GetObject([pot] [, razred])
```

Prvi parameter je neobvezen in ga podamo, ko bi radi vzpostavili referenco na obstoječi predmet, shranjen v datoteki. Če za pot zapišemo prazen niz, se funkcija vede enako kakor funkcija CreateObject. Preglednico, ki smo jo shranili v prvem zgledu tega poglavja, lahko predstavimo kot predmet takole:

```
Dim xlSheet As Object
  Set xlSheet = GetObject("c:\Zvezek.xls")
```

Uporaba vmesnika

Po vzpostavljanju predmeta lahko brez omejitev uporabljamo njegove lastnosti in metode, kakor da bi bil vgrajen razred visual basica. Če želimo, lahko vzpostavimo tudi odziv na dogodke predmeta. To je mogoče le za razrede, ki smo jih vključili v projekt z njihovo razredno knjižnico.

Povsem enako, kakor velja za lastne razrede, napovemo predmetno spremenljivko z dogodki z določilom WithEvents. Po taki napovedi bodo novi dogodki takoj na voljo v oknu s kodo, na ustreznem seznamu desno zgoraj.

Oglejmo si zgled, ki prepreči zapiranje okna v Excelu:

```
Private WithEvents xlBook As Excel.Workbook

Private Sub Form_Load()
  Set xlBook = GetObject("c:\Zvezek.xls")
  xlBook.Application.Visible = True
  xlBook.Windows(1).Visible = True
End Sub

Private Sub xlBook_BeforeClose(Cancel As Boolean)
  xlBook.Application.Visible = False
  MsgBox "Ta delovni zvezek nujno potrebujem!"
  xlBook.Application.Visible = True
  Cancel = True
End Sub
```

Najprej napovemo zasebno predmetno spremenljivko, ki bo prožila dogodke. Pri vzpostavitvi obrazca poskrbimo tudi za vzpostavitev predmetne spremenljivke in z njenimi lastnostmi prikažemo gostitelja, program Microsoft Excel. Če bomo v Excelu poizkusili zapreti prikazani delovni zvezek ali morda kar ves program, bo povezava prek mehanizma dogodkov najprej skrila gostitelja, prikazala opozorilo, zatem znova prikazala gostitelja in preprečila nadaljnje izvajanje dogodka. Tako bi lahko zavarovali pomembne podatke v delovnem zvezku, nujno potrebne pri nadaljevanju dela.

Sproščanje

Podobno, kakor pri uporabi kateregakoli drugega razreda velja, da se ta dokončno odstrani iz pomnilnika šele, ko nanj ne kaže nobena predmetna spremenljivka več. Zaradi povezanosti razredov v večini razrednih knjižnic je nujno, da ob koncu dela sprostimo vsa sklicevanja, tudi na podrejene razrede. Zgled lahko prepišemo iz prvega zgleda v tem poglavju:

```
  ' Sprostitev referenc
  Set xlSheet = Nothing
  Set xlBook = Nothing
  Set xlApp = Nothing
```

Obravnavanje izvajalnih napak

Pri uporabi zunanjih izpostavljenih razredov je zelo pomembno učinkovito prestrezanje izvajalnih napak. Pogosto se pripeti, da je uporabnik hote ali nehote zaprl strežni program in s tem onemogočil nadaljevanje dela. Tovrstne napake ne moremo zaznati z ugotavljanjem pravilnosti predmetne reference. Izraz

predmet **Is Nothing**

vrne resnico le pri nevzpostavljeni predmetni spremenljivki, ne more pa ugotoviti, ali je referenca vzpostavljene spremenljivke vmes postala neveljavna (npr. zaradi zgoraj opisanega vzroka). Preostane nam le, da vzpostavimo prestrezovalnik izvajalnih napak. Ponazarja ga naslednji zgled:

```
Private xlApp As Excel.Application

Private Sub Command1_Click()
  StartXL
End Sub

Private Sub StartXL()
Const AUTOMATION_ERROR = 440
Dim intZnova As Integer

  intZnova = 0

On Error GoTo Napaka
  Set xlApp = New Excel.Application
  xlApp.Visible = True
  Exit Sub

Napaka:
  Select Case Err.Number
    Case AUTOMATION_ERROR
      intZnova = intZnova + 1
      If intZnova < 5 Then
        Resume
      Else
        Err.Raise Number:=vbObjectError + 7001, _
          Description:="Ne morem zagnati Excela"
      End If
    Case Else
      Err.Raise Number:=Err.Number
  End Select

End Sub
```

Tako zapisan prestrezovalnik ni najuporabnejši pri predmetih, ki lahko povzročijo izvajalne napake s kodami, ki niso združljive s kodami, ki veljajo v visual basicu. V tem primeru nam ne preostane drugega, kakor da napake zanemarimo in jih preverjamo s pogojnim stavkom po vsakem ukazu, ki bi jih lahko povzročil:

```
On Error Resume Next

  ' ...

  Set xlApp = New Excel.Application

  If Err.Number <> 0 Then

    ' ...

  End If
```

Prezaposlenost

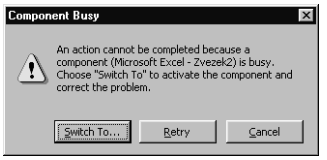
Klic metode ali nastavitev lastnosti terja le kratek čas, njeno obravnavanje pa se včasih lahko zavleče. To se lahko zgodi v naslednjih primerih:

- Zahtevamo storitev od strežnika, ki pa prikazuje modalno pogovorno okno in čaka odziv uporabnika, zato nas ne more postreči. Temu stanju pravimo *zaposleni gradnik* (*component busy*).
- Medtem ko se še vedno izvaja prejšnja zahteva, npr. dolgotrajna poizvedba v zbirki podatkov, izdamo novo zahtevo. Temu stanju pravimo *čakajoča zahteva* (*request pending*).
- Dva ali več programov zahteva storitev od deljenega gradnika. Zaradi zaporednega izvrševanja (*serialization*) se druga zahteva ne more izvršiti, dokler je v izvajanju prva. Tudi to pripelje do čakajoče zahteve.

Zaposleni gradnik

Če je strežnik zaposlen, bo visual basic določen časovni interval ponavljal zahteve po njegovi storitvi. Če se po izteku tega intervala strežnik še vedno ne odzove, visual basic prikaže posebno pogovorno okno, ki nam omogoča preklop v strežnik, v katerem morda lahko z ročno intervencijo odpravimo vzrok čakanja. To pogovorno okno smo že opazili pri prikazu modalnih oken zunajprocesnih strežnikov. Vidimo ga tudi na spodnji sliki:

Slika 20-2: Sporočilo o zaposlenem gradniku



Če nam težave ne uspe odpraviti, lahko v pogovornem oknu izberemo gumb Cancel, s katerim se nadaljnje poizkušanje prekine, kličoči podprogram pa doleti izvajalna napaka.

Čakajoča zahteva

Če strežnik sprejme zahtevo, mora kličoči program – odjemalec – počakati, da se zahteva izvrši. Če želi med tem uporabnik kaj postoriti z uporabniškim vmesnikom odjemalca, npr. minimizirati obrazec, bo visual basic po kratkem premoru prikazal posebno pogovorno okno, podobno obvestilu o zaposlenem gradniku, le da je v tem oknu onemogočen gumb Cancel, saj izvajajoče se zahteve ni mogoče več preklicati. Tudi tu lahko pomaga preklop v strežnik, saj je vzrok čakanja lahko npr. modalno prikazano pogovorno okno s sporočilom o napaki.

Lastna sporočila

Privzeta pogovorna okna z obvestili o zaposlenem strežniku marsikdaj niso uporabna in lahko zmedejo uporabnika. Na srečo jih lahko nadomestimo z običajnim pogovornim oknom, kot ga prikaže vgrajena funkcija MsgBox. V tem primeru moramo nastaviti nekaj posebnih lastnosti vgrajenega predmeta App:

Lastnost	Opis
OLEServerBusyMsgText	Določa besedilo, ki se prikaže v pogovornem oknu sporočila o zaposlenem strežniku. Običajno pogovorno okno, ki nadomesti privzetega, ima gumb OK, ki prevzame vlogo gumba za ponovitev zahteve, in gumb Cancel, ki prekliče nadaljnje poizkuse.
OLEServerBusyMsgTitle	Določa naslov okna s sporočilom o zaposlenem strežniku. Če nastavimo samo to lastnost, se še vedno prikaže privzeto pogovorno okno visual basica.
OLERequestPendingMsgText	Določa besedilo v pogovornem oknu sporočila o čakajoči zahtevi. Običajno pogovorno okno, ki nadomesti privzetega, ima le gumb OK.
OLERequestPendingMsgTitle	Določa naslov okna s sporočilom o čakajoči zahtevi. Če nastavimo samo to lastnost, se še vedno prikaže privzeto pogovorno okno visual basica.

S posebnimi lastnostmi vgrajenega predmeta App lahko nadzorujemo tudi časovni interval ponavljanja zahtev oziroma odziva na naknadno povzročene dogodke pri čakajoči zahtevi. Ti dve lastnosti sta:

Lastnost	Opis
OLEServerBusyTimeout	Časovni interval v milisekundah, v katerem visual basic ponavlja zahteve pri zaposlenem strežniku. Privzeta nastavitev je 10.000, kar pomeni 10 sekund.
OLERequestPendingTimeout	Časovni interval v milisekundah, po katerem se visual basic odzove na naknadno povzročene dogodke pri čakajoči zahtevi. Privzeta nastavitev je 5000 (5 sekund).

Najkoristnejša lastnost pa je verjetno **OLEServerBusyRaiseError**. Če jo nastavimo na resnično vrednost, se namesto prikaza pogovornega okna sproži posebna izvajalna napaka, ki jo v programski kodi lahko prestrežemo in najprimerneje obdelamo.

Vizualno delo

Pri obravnavi vezanih gradnikov v tretjem delu knjige smo že dodobra spoznali poseben vgrajeni gradnik razreda OLE, ki nam omogoča izkoriščati izpostavljene razrede brez uporabe programske kode. V trenutku, ko izvod omenjenega gradnika postavimo na uporabniški vmesnik, se prikaže pogovorno okno, s katerim določimo *vstavljeni* oziroma *povezani* predmet, ki ga bo gradnik prikazoval.

Ta predmet lahko med izvajanjem programa sprožimo in uporabljamo njegove zmožnosti, ki jih priskrbi strežnik v ozadju. Natančnejši opis njegovih lastnosti in metod najdete v razdelku Zmogljivi vezani gradniki v 11. poglavju.

21. POGLAVJE

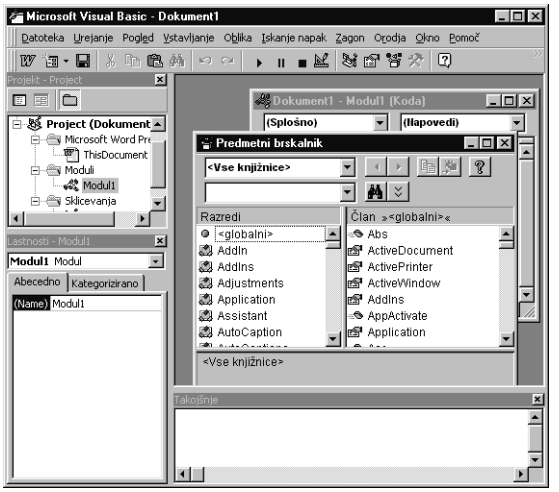
Narečja

V tem poglavju si bomo ogledali dve razprostranjeni različici basica. Obe sta okrnjeni različici jezika, ki je na voljo v razvojem okolju visual basic. To sta visual basic for applications, namenjen predvsem nadziranju zmogljivih namenskih programov, in visual basic scripting, ki je namenjen nadziranju manj zmogljivih programov in sistemskemu upravljanju.

VBA

Visual basic for applications (VBA) je temeljna razredna knjižnica, ki opredeljuje narečje programskega jezika basic, uporabljeno v visual basicu. Microsoft se je odločil, da tehnologije[68] VBA ne bo uporabil le v posebnem razvojnem orodju in v vseh izdelkih iz skupka Office, ampak jo je dal naprodaj. V kratkem lahko pričakujemo precej namenskih programov, ki bodo za svoj nadzor ponujali programiranje v VBA.

Slika 21-1: Razvojno okolje VBA v namenskem programu



[68] Izraz »Tehnologija VBA« predstavlja temeljno razredno knjižnico in posebno obliko razvojnega okolja za njeno rabo.

Seveda je VBA le okostje jezika, ki mu je treba dodati še zmogljivo razredno knjižnico. Ker so novejši programi tako ali tako večinoma pripravljeni kot samostojni strežniki ActiveX, to več ni nobena težava. Seveda je širitev visual basica voda na mlin vsem razvijalcem, ki ga uporabljajo, saj se hkrati močno povečuje tudi trg za najrazličnejše dodatke, ki jih lahko ustvarimo z gradniki ActiveX. Tako kakor smo v prejšnjem poglavju uporabljali izpostavljene razrede drugih programov, lahko ti uporabljajo izpostavljene predmete naših strežnikov.

NAMIG Ni nujno, da drugi programi uporabljajo prav visual basic. Gradniki ActiveX so del specifikacije COM (*Component Object Model*), ki omogoča souporabo gradnikov, razvitih z najrazličnejšimi orodji. Zadošča, da vsi izpeljejo standardizirane vmesnike, in že je mogoče njihovo medsebojno komuniciranje. Gradnike, razvite v visual basicu, lahko uporabimo v razvojnih orodjih, kot sta npr. Microsoft Visual C++ ali Borland Delphi.

Pisarna

Znanje, ki smo ga pridobili v prvih dveh delih te knjige, lahko brez večjih težav uporabimo pri vseh izdelkih Microsoftove zbirke Office, ki podpirajo programiranje v VBA. Skladnja jezika je v vseh izdelkih enaka, spreminja se le poseben predmetni model, ki predstavlja notranji ustroj posameznega programa. Tudi razvojno okolje je skorajda enako visual basicu (glej sliko 21-1).

Ker večina orodij poleg ročnega kodiranja podpira tudi samodejno pretvarjanje zaporedja ukazov v ustrezne stavke VBA, je spoznavanje uporabe predmetnega modela toliko lažje. Za začetek preprosto sprožimo »snemanje makro ukaza« in interaktivno izvedemo zanimivo zaporedje ukazov. Po koncu »snemanja« se lahko preselimo v urejevalnik kode in jo tam ročno posodobimo.

VBS

Visual basic scripting (VBS) je več kot samo okrnjena različica narečja basica. Microsoft je skriptno orodje zasnoval mnogo širše. Vsi ciljni programi, ki jih je mogoče nadzorovati z VBS, pravzaprav uporabljajo tehnologijo ActiveX Scripting, ki omogoča uporabo poljubnega skriptnega jezika. ActiveX Scripting poskrbi za prevajanje kode iz poljubnega skriptnega jezika v enakovredno vmesno obliko, ki jo želi Microsoft standardizirati in s tem odpreti vrata najrazličnejšim skriptnim jezikom. Zaenkrat sam podpira uporabo VBS in prilagojenega narečja skriptnega jezika javascript (v Microsoftovem žargonu imenovan jscript). Ker v tej knjigi govorimo o visual basicu, se bomo v nadaljevanju osredotočili zgolj na uporabo VBS.

Elementi jezika

V programski kodi VBS lahko uporabljamo en sam podatkovni tip: prilagodljivi tip Variant. To je hkrati tudi edini podatkovni tip, ki ga lahko vrača funkcija. Funkcionalnost jezika ni zaradi tega prav nič okrnjena, saj lahko prilagodljivi tip vsebuje kakršenkoli podatek, tudi referenco predmeta ali oznako napake. VBS podpira večino funkcij za pretvorbo tipa (spoznali smo jih v prvem delu knjige), ki lahko iz podatka tipa Variant izluščijo najrazličnejšo vsebino.

Uporabimo lahko poljubno število spremenljivk. Zanje veljajo enaka pravila poimenovanja kakor v visual basicu. Če želimo, lahko skriptno kodo začnemo s stavkom Option Explicit in s tem zahtevamo predhodno napoved vseh uporabljenih spremenljivk. Spremenljivke lahko napovedujemo s stavki Dim, Private ali Public. Pri njihovem dosegu in življenjski dobi sta pri VBS le dve pravili:

- Spremenljivke, napovedane znotraj podprogramov, so vidne in živijo le znotraj njega (*procedure-level variables*).
- Spremenljivke, napovedane zunaj podprogramov, so na voljo v vsej skripti (*script-level variables*).

Polja v VBS imajo spodnjo mejo intervala zaporednih števil vedno nastavljeno na 0.

Uporaba konstant ter zapis konstantnih vrednosti se v VBS ne razlikuje od visual basica. VBS pozna tudi nekaj vgrajenih konstant. Omenimo konstanto vbNewLine, ki predstavlja platformsko odvisno označitev prehoda v novo vrsto. V določenih sistemih je vrednost te konstante znak s kodo ASCII 10, drugje pa par znakov s kodama 13 in 10.

VBS podpira večino operatorjev visual basica, izjema je operator Like za primerjanje niza z vzorcem in operator TypeOf za ugotavljanje razreda. VBS tudi ne podpira operatorja ! (*bang*) za hitrejše doseganje elementov zbirke.

Na voljo so vse strukture za nadziranje izvajanja programa, kot jih poznamo iz visual basica: od vejitvenih stavkov If ... Then ... Else in Select Case do zank Do, For in For Each.

Posebej poenostavljeno je nadzorovanje napak. Na voljo sta samo stavka On Error Resume Next, ki prezre izvajalne napake, in stavek On Error Goto 0, ki izklopi tako vedenje. Zato lahko ugotavljamo morebitno izvajalno napako le s preverjanjem lastnosti predmeta Err.

VBS omogoča uporabo vgrajenih funkcij MsgBox in InputBox, ki ju uporabimo za prikaz najosnovnejših oblik pogovornih oken. Lastnih obrazcev ne moremo določiti neposredno, lahko pa izkoristimo posebej pripravljen gradnik ActiveX.

V VBS ne moremo uporabljati običajnih funkcij za nizkonivojsko delo z datotekami in datotečnim sistemom in ne moremo napovedovati zunanjih funkcij. Lahko pa uporabljamo strežnike ActiveX. Ravno uporaba zunanjih strežnikov prek avtomatizacije praktično neomejeno širi zmogljivosti VBS.

Pripravljeni razredi

VBS pozna nekaj vgrajenih predmetov in nekaj razredov, ki jih lahko uporabimo v posebne namene. To so:

Predmet/Razred	Opis
Err	Nam že dobro znani predmet z opisom zadnje napake in metoda, ki omogoča proženje lastnih napak. To je tudi edini vgrajeni predmet, katerega izvod je že vzpostavljen.
Dictionary	Razred, ki določa podatkovno strukturo *slovarja*. Slovar ali asociativno polje omogoča shranjevanje povezanih parov (ključ : vrednost). Omogoča hitro iskanje po vsebini.
FileSystemObject	Razred, ki omogoča doseg datotečnega sistema. Z njegovimi metodami lahko ustvarimo ali odpremo datoteko z besedilom.
TextStream	Razred, ki predstavlja ustvarjeno ali odprto datoteko z besedilom. Njeno vsebino predstavi kot *tok* (*stream*), iz katerega lahko črpamo ali vpisujemo vsebino.

Vsi našteti razredi (razen vgrajenega predmeta Err) so v razredni knjižnici Scripting in jih lahko ustvarjamo s klicem funkcije CreateObject. Naslednji zgled ustvari datoteko z besedilom in vanjo zapiše vrstico:

```
Set fileSys = CreateObject("Scripting.FileSystemObject")
Set txtFile = fileSys.CreateTextFile("c:\testfile.txt", True)
txtFile.WriteLine("Pozdravljen, datotečni svet.")
txtFile.Close
```

Obe uporabljeni spremenljivki sta bili napovedani implicitno. Če želimo doseči večjo trdoživost kode, je koristno v prvo vrstico zapisati stavek Option Explicit. Prednosti njegove uporabe smo že spoznali.

Upravljanje

Na priloženi plošči CD-ROM najdemo dodatek Windows Scripting Host, ki omogoči uporabo VBS za sistemsko upravljanje (namestitev je opisana v dodatku). Z njim lahko nadziramo register, upravljamo datotečni sistem in druga sistemska sredstva. Ker sam VBS vseh teh opravil ne podpira, pri datotečnem sistemu nam je npr. na voljo zgolj ustvarjanje in branje tekstovnih datotek, je Windows Scripting Host popestren s posebno razredno knjižnico, ki vključuje vse potrebne razrede za upravljanje sistemskih sredstev.

Za prikaz uporabe si oglejmo razmeroma preprosto skripto, ki ustvari novo bližnjico na namizju. Kodo zapišemo v datoteko, ki ima končnico VBS. Če imamo pravilno

nameščen dodatek Windows Scripting Host, je mogoče tovrstne skripte pognati neposredno, bodisi z dvakratnim klikom imena datoteke v Raziskovalcu bodisi z vnosom imena datoteke v konzolno okno (podobno, kakor lahko poženemo datoteke s podaljškom BAT).

```
' Prikaz uporabe VBS skozi WSH za ustvarjanje nove bližnjice na namizju

Option Explicit

' Pozdrav in vprašanje na začetku
Sub Pozdrav()
  If MsgBox("Pozdravljeni v prikazu zmogljivosti WSH " & vbLf & _
       "Želite nadaljevati s stvaritvijo nove bližnjice?", _
      vbOKCancel + vbInformation, "Prikaz WSH" )= vbCancel Then
    WScript.Quit
  End If
End Sub

' Izvajanje se začne tu
Pozdrav

' Če podprogram ne prekine izvajanja, nadaljujemo
Dim WSHShell
Dim MyShortcut, MyDesktop, DesktopPath

' Poseben predmet za dostop do Raziskovalca
Set WSHShell = WScript.CreateObject("WScript.Shell")

' Preberemo pot do posebne mape
DesktopPath = WSHShell.SpecialFolders("Desktop")

' V tej mapi ustvarimo novo bližnjico
Set MyShortcut = WSHShell.CreateShortcut(DesktopPath & _
"\Bližnjica do Beležke.lnk")

' Priredimo ustrezne lastnosti novi bližnjici
MyShortcut.TargetPath = _
  WSHShell.ExpandEnvironmentStrings("%windir%\notepad.exe")
MyShortcut.WorkingDirectory = WSHShell.ExpandEnvironmentStrings("%windir%")
MyShortcut.WindowStyle = 4
MyShortcut.IconLocation = _
  WSHShell.ExpandEnvironmentStrings("%windir%\notepad.exe, 0")

' Shranimo bližnjico
MyShortcut.Save

' Končno obvestilo
WScript.Echo "Nova bližnjica do beležke je bila dodana na namizje"
```

Kakor vidimo, večino dela opravijo pripravljeni razredi v knjižnici WScript, vse skupaj pa povezuje koda v nam prav dobro znanem narečju basica.

NAMIG Razredno knjižnico Windows Scripting Host (WSH) lahko vključimo tudi v projekte visual basica. Kodo zgleda najdete na priloženi plošči CD-ROM v mapi \Primeri\5-Svet\2-Narecja\. Opis namestitve sistemskega dodatka WSH je na koncu knjige.

Spletni programi

Zaradi nove strateške usmeritve podjetja Microsoft, da bi čimbolj obvladovali internet, nas to podjetje zasipava z brezplačnimi darili. Začelo je z brezplačnim brskalnikom, sledil mu je nadvse zmogljiv spletni strežnik, ki je prav tako brezplačen in ga najdete tudi na priloženi plošči CD-ROM. Spletni strežnik Internet Information Server (IIS) in njegove dejavne strežene strani (ASP) so še ena izmed uporab jezika VBS. Tudi strežnik IIS uporablja tehnologijo ActiveX Scripting, tako da je VBS le eden izmed podprtih jezikov.

NAMIG Opis postopka namestitve spletnega strežnika najdete v dodatku na koncu knjige.

Dejavne strani

Za uporabo na dejavnih straneh ASP so VBS še malo okrnili. Odvzeli so mu funkciji InputBox in MsgBox, saj spletna stran za komuniciranje z uporabnikom uporablja čisto drugačne mehanizme. Po drugi strani so jezik izdatno popestrili z več razrednimi knjižnicami, od katerih je najbolj impresivna knjižnica ActiveX Data Objects (ADO). Ta pokriva praktično vso zmogljivost predmetnega modela DAO, ki smo ga spoznali v tretjem delu knjige, s tem da je posebej prilagojena čimbolj učinkoviti rabi na dejavnih spletnih straneh, ki jih lahko hkrati pregleduje zelo veliko uporabnikov.

Preprost zgled

Dejavne strežene strani ustvarimo tako, da na spletno stran, ki je kodirana v označevalnem jeziku HTML, vpletemo kodo v VBS. To kodo strežnik prepozna in smiselno uporabi, prek interneta pa se posreduje le končni izid, ki ne vsebuje več nobene kode v VBS. Skriptni ukazi na strani strežnika so zato skriti očem odjemalca. Oglejmo si zgled zelo preproste dejavne spletne strani:

```
<HTML>
<BODY>
<% For i = 4 To 7 %>
<FONT SIZE=<% = i %>>
Pozdravljen, svet!<BR>
<% Next %>
</BODY>
</HTML>
```

Koda v VBS je omejena s posebnima oznakama, <% in %>, in se po tem razlikuje od vseh oznak jezika HTML. Če to vsebino shranimo v datoteko s podaljškom ASP in jo posredujemo v svet prek spletnega strežnika IIS, bo odjemalec prejel naslednji izid:

```
<HTML>
<BODY>
<FONT SIZE=4>
Pozdravljen, svet!<BR>
<FONT SIZE=5>
Pozdravljen, svet!<BR>
<FONT SIZE=6>
Pozdravljen, svet!<BR>
<FONT SIZE=7>
Pozdravljen, svet!<BR>
</BODY>
</HTML>
```

Koda v VBS je povzročila štirikratno ponovitev omejenega segmenta spletne strani, ki je bil ujet v zanki For Next. Kot vidimo, odjemalec sploh ne more vedeti, ali je vsebina strani izid statične kode ali je ustvarjena s programom.

Pogovor med stranmi

Velika težava pri uporabi spletnih strani, ki se prenašajo po protokolu HTTP (*Hyper Text Transfer Protocol*), je v njegovi pozabljivosti. Protokol za prenos vsake vsebine vzpostavi povezavo med odjemalcem in strežnikom in jo takoj po prenosu poruši. Ob vnovičnem obisku tako ne moremo vedeti, ali nas je ta obiskovalec že obiskal. No, na srečo so na voljo posebni mehanizmi, kot je prenos parametrov v naslovu URL ali uporaba piškotkov (*persistent state cookies*), ki pa jih ne podpirajo čisto vsi odjemalci.

Dejavne strežene strani vpeljujejo celo vrsto predmetov, ki komunikacijo med stranmi zelo poenostavijo. Ti predmeti so zbrani v spodnji tabeli:

Predmet	Opis
Server	Nadzoruje okolje izvajanja dejavnih strani.
Request	Omogoča branje podatkov z več zbirkami, ki so njegove lastnosti. Te zbirke opisujejo strežnik, posredovane parametre v naslovu URL, skrite posredovane parametre, piškotke in odjemalčeve certifikate.
Response	Omogoča pošiljanje podatkov. Pozna metode za izpis vsebine na spletno stran, preusmeritev odjemalca na novo stran in nastavitev piškotkov.
Session	Omogoča shranjevanje posebnih podatkov za vsakega posameznega obiskovalca.
Application	Omogoča shranjevanje globalnih podatkov, vidnih vsem obiskovalcem.

Oglejmo si zgled malo bolj zapletene spletne strani, ki omogoča odziv na vnos uporabnika v obrazec. Izbrano vrednost hkrati tudi shrani v predmet Session, tako da bo v nadaljevanju znana vsem drugim dejavnim stranem:

```
<!-- stran z imenom zgled.asp -->
<HTML>

<HEAD>
  <TITLE>Zgled uporabe dejavnih strani</TITLE>
</HEAD>

<BODY>

  <% If IsEmpty(Request("izbor")) Then %>

     <FORM ACTION="zgled.asp" METHOD="post">
     Izberi najljubše razvojno orodje:
     <SELECT NAME="izbor">
       <OPTION VALUE="Visual Basic">Visual Basic
       <OPTION VALUE="Delphi">Delphi
       <OPTION VALUE="JBuilder">JBuilder
        <OPTION VALUE="Visual C++">Visual C++
      </SELECT>
     <INPUT TYPE="Submit" VALUE=" Potrdi ">
   </FORM>

  <%
     Else

      Session("izbor") = Request("izbor")
      Response.Write("<H1>Izbral si " & Session("izbor") & "</H1>")

     End If
  %>

</BODY>
</HTML>
```

Stran preveri vrednost parametra »izbor«. Če ni določen, prikaže obrazec, ki bo isto stran poklical znova (neke vrste rekurzivni klic), tokrat z ustrezno nastavljenim parametrom, ki je posledica izpolnjenega obrazca. Kot vidimo, je vsa skrivnost v poznavanju novih razredov (in poznavanju HTML[69]). Uporabljeni jezik je spet nam zelo dobro znano narečje basica.

Predmeti ADO

Za dostop do zbirk podatkov je spletnemu strežniku dodana razredna knjižnica ActiveX Data Objects (ADO). Večina razredov v knjižnici ADO podpira neposredno ustvarjanje predmetov, brez zapletenega brskanja po predmetnem modelu, ki smo mu bili priča pri uporabi modela DAO. S tem se izdatno zmanjša potrebni pomnilnik in poveča učinkovitost strežbe.

Drugače je raba predmetov modela ADO zelo podobna rabi predmetnega modela DAO, ki smo ga spoznali v tretjem delu knjige. Oglejmo si preprost zgled, ki opravi poljubno poizvedbo, vneseno v obrazec spletne strani, in izid prikaže v obliki tabele:

[69] Če vas podrobneje zanima protokol HTTP, označevalni jezik HTML, in programiranje v javi ali javascriptu, si oglejte knjigo iz zbirke Monitor (v kateri je izšla tudi ta knjiga): Java, programiranje za Internet.

```
<!-- datoteka poizvedba.asp -->
<HTML>
<HEAD>
  <TITLE>Prikaz izida poljubne poizvedbe</TITLE>
</HEAD>

<BODY>
<%
  vntSQL = Request("Q")
  If IsEmpty(vntSQL) Then
%>
    <FORM ACTION="poizvedba.asp" METHOD="get">
    <INPUT NAME="Q" SIZE=100>
    </FORM>
<%
  Else
%>
    <FORM ACTION="poizveba.asp" METHOD="get">
    <INPUT NAME="Q" SIZE=100 VALUE="<% = vntSQL %>">
    </FORM>
<%
    Set Conn = Server.CreateObject("ADODB.Connection")
    Conn.Open "DSN=VirODBC;UID=sa;PWD=;"
    Set Result = Conn.Execute(vntSQL)
%>

    <TABLE BORDER=1 CELLSPACING=0 CELLPADDING=5>
<%
    Number = 0
    For Each field In Result.Fields
      Response.Write("<TH BGCOLOR=silver>")
      Response.Write(field.name)
    Next
    Do While Not Result.EOF
      Number = Number + 1
      Response.Write("<TR>")
      For Each field In Result.Fields
        Response.Write("<TD>")
        Response.Write(field.value)
      Next
      Result.MoveNext
    Loop
%>
    </TABLE>

<%
    Result.Close
    Conn.Close
  End If
%>

</BODY>
</HTML>
```

Funkcija GetObject se pri dejavnih straneh preslika v metodo razreda Server. Opazimo lahko, da je delo s predmeti modela ADO zelo podobno rabi modela DAO.

Globalna datoteka

Kadar načrtujemo zmogljivejše spletne programe, jim v spletnem strežniku dodelimo posebno navidezno mapo, ki kaže na neko fizično pot v datotečnem sistemu. Če je na tej poti posebna datoteka, imenovana Global.ASA, jo spletni strežnik uporabi za vzpostavitev in zaustavitev programa, njeno programsko kodo pa uporabi tudi ob prihodu in odhodu vsakega novega obiskovalca. Datoteka Global.ASA vsebuje kodo v VBS, ki je lahko vključena v štiri posebne odzivne procedure. To so:

Odzivna procedura	Opis
Application_OnStart	Procedura se pokliče ob vzpostavitvi spletnega programa.
Application_OnEnd	Procedura se pokliče ob zaustavitvi spletnega programa.
Session_OnStart	Procedura se pokliče ob prihodu novega obiskovalca.
Session_OnEnd	Procedura se pokliče po daljšem času mirovanja obiskovalca (kar se razume kot njegov odhod).

V preostali kodi znotraj datoteke Global.ASA lahko ustvarimo izvode predmetov, ki jih bomo zelo pogosto uporabljali v vsem spletnem programu. Oglejmo si zgled, ki ustvari globalno dosegljiv izvod predmeta za delo z datotečnim sistemom, v odzivnih procedurah pa skrbi za števec obiskovalcev:

```
' Vzpostavitev globalno dosegljivega predmeta
<OBJECT RUNAT=Server SCOPE=Application ID=FileSys
  PROGID="Scripting.FileSystemObject">
</OBJECT>

<SCRIPT RUNAT=Server Language=VBScript>
Sub Application_OnStart
  ' Ob vzpostavitvi strežnika vzpostavimo tudi
  ' števec obiskov s pomožno datoteko
  fName = Server.MapPath ("/") + "\stevec_obiskov.txt"
  On Error Resume Next
  Set FObject = FileSys.OpenTextFile (fName, 1, FALSE, FALSE)
  Application("Stevec") = FObject.ReadLine
  On Error Goto 0
  If Err Then
    Application("Stevec") = 0
  End If
  Application("Datoteka") = fName
End Sub

Sub Application_OnEnd
  ' Če pride do zaustavitve strežnika, se število obiskov zapiše
  ' v pomožno datoteko
  Set FObject = FileSys.CreateTextFile (Application("Datoteka"), _
    TRUE, FALSE)
```

```
  FObject.WriteLine(Application("Stevec"))
  FObject.Close
End Sub

Sub Session_OnStart
  ' Ob vsakem novem obisku povečamo števec
  Application.Lock
  Application("Stevec") = Application("Stevec") + 1
  Application.Unlock

  ' Na vsakih 20 obiskov stevec za vsak primer shranimo v datoteko
  If Application("Stevec") Mod 20 = 0 Then
    Set FObject = FileSys.CreateTextFile (Application("Datoteka"), _
      TRUE, FALSE)
    Application.Lock
    FObject.WriteLine(Application("Stevec"))
    FObject.Close
    Application.Unlock
  End If
End Sub
```

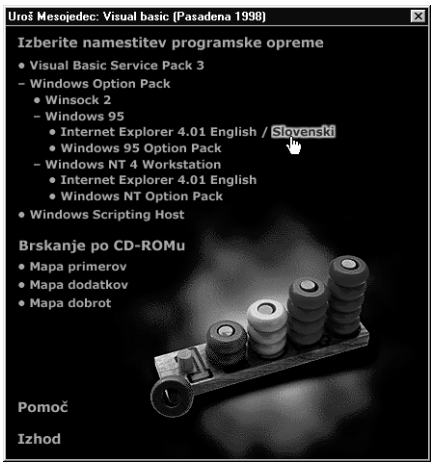
Iz zgleda lahko najprej razberemo alternativno možnost za ustvarjanje predmetov, ki je mogoča tudi s posebno oznako jezika HTML. Naslednja zgledna tehnika je prikaz lovljenja izvajalne napake z omejenimi zmožnostmi VBS. Zgled prav tako prikazuje metode za posodabljanje stanja predmeta Application. Ker ima do predmeta lahko posredno dostop tudi več obiskovalcev hkrati, je nujno, da ga pred posodabljanjem zaklenemo in po posodobitvi spet odklenemo.

DODATEK A

Priložena plošča

Knjigi je priložena plošča CD-ROM z izvorno kodo zgledov iz knjige in nekaj uporabnimi datotekami. Če imate na voljo samodejni zagon, zadošča, da ploščo vstavite v pogon. Drugače ročno poženite program Arunvb.EXE, ki prikaže uvodni menu, prek katerega boste najlaže namestili dodatno programsko opremo. Sliko uvodnega menuja si lahko ogledate spodaj.

Slika A-1: Uvodni menu plošče CD-ROM



Če želite namestiti katerega izmed naštetih dodatkov za Okna, ga z miško kliknite. Naslednje dodatke je mogoče namestiti v Okna 95 ali NT:

Dodatek	Opis
Visual Basic Service Pack 3	Popravki za razvojno okolje visual basic. Dodatek obsega tudi celoten Service Pack 2, zato ga lahko namestimo neposredno prek prve namestitve visual basica.
Winsock 2	Izboljšava vmesnika Winsock, ki jo moramo namestiti, če želimo uporabljati dodatek Option Pack.
Windows Scripting Host	Dodatek za Okna, ki omogoča pisanje sistemskih skript.

Naslednji dodatki pa so na voljo v dveh različicah, eni za Okna 95, drugi pa za Okna NT 4 (Workstation):

Dodatek	Opis
Internet Explorer 4.01	Spletni brskalnik, ki je za Okna 95 na voljo v angleški in slovenski različici. Nujno ga je namestiti, če želimo uporabiti dodatek Option Pack.
Option Pack	Dodatki za operacijski sistem, med katerimi najdemo spletni strežnik Internet Information Server 4.

NAMIG Uvodno kazalo plošče CD-ROM ponuja kratko pomoč, ki jo dosežete s pritiskom na tipko F1. Če vas kazalo ne zanima več, ga lahko hitro zaprete s pritiskom na tipko Esc.

Uvodno kazalo ponuja še bližnjice do najpomembnejših map plošče CD-ROM. Opis vsebine večine map je v spodnji tabeli:

Mapa	Vsebina
Primeri	V mapi so datoteke zgledov iz knjige. Večina zgledov je na voljo v izvorni kodi in tudi v prevodu.
Vbsp3	Datoteke popravkov razvojnega okolja Visual Basic Service Pack 3.
Dodatki	Najrazličnejši dodatki za operacijske sisteme Okna. Poleg zgoraj naštetih najdemo v tej mapi tudi dodatke Windows NT Service Pack 3, DirectX 5 in Active Movie (eden izmed podsklopov, ki se namestijo tudi skupaj s spletnim brskalnikom Internet Explorer).
Dobrote	Nekaj izbranih posladkov, kot so novi zunanji gradniki, izvajalne knjižnice vseh doslej izdanih različic visual basica ter nekaj uporabnih programov.

DODATEK B

Opis datotek

Razvojno okolje visual basica in nekatera pomožna orodja (npr. Application Setup Wizard) uspešno združujejo cel kup najrazličnejših datotek v projekte ali celo skupine projektov. V tem dodatku si bomo pregledno ogledali namen in kratek opis vsebine pomembnejših vrst datotek. Za njihovo urejanje je večinoma najprimernejše orodje seveda kar sam visual basic, vendar včasih lahko kaj postorimo tudi zunaj njega, predvsem pa je poznavanje vrst datotek pomembno pri prenašanju izvorne kode iz računalnika v računalnik. Naj še enkrat poudarim, da je najbolje vsak, tudi povsem preprost projekt shraniti v posebno mapo.

V nadaljevanju bomo videli, da je večina datotek shranjenih kar kot navadno besedilo, ki spominja na zapisa datotek .INI, ki jih je v novejših Oknih izpodrinil register (*registry*). Zato je večino datotek mogoče pregledovati in celo popravljati tudi zunaj razvojnega okolja, kar pa priporočam samo bolj izkušenim uporabnikom. Pred kakršnimkoli posegom v datoteko zunaj razvojnega okolja si nujno pripravite rezervno kopijo.

Pri opisu vsebine datotek bomo uporabili simbolično oznako elementov, ki jo zapišemo med lomljenima oklepajema (<>).

Skupina projektov

Te datoteke imajo podaljšek .VBG. V njih je navadno besedilo, ki določa seznam projektov v skupini. Najpogostejša oblika datoteke je:

```
VBGROUP <različica>
StartupProject=<projekt1>.vbp  ' zagonski projekt
Project=<projekt2>.vbp         ' projekt v skupini
```

Projekt

Posamezen projekt je shranjen v datoteki s podaljškom .VBP. Tudi opis projekta je navadno besedilo, katerega najpogostejša oblika se glasi:

```
Type=Exe                              ' vrsta projekta
Object=<GUID>#<verzija>#<datoteka>    ' vključen zunanji gradnik
```

```
Reference=<GUID>#<verzija>#<datoteka> ' vključena razredna knjižnica
Class=<ime razreda>; <razred1>.cls    ' lasten razred projekta
Module=<ime modula>; <modul1>.bas     ' modul s programsko kodo
Form=<obrazec1>.frm                   ' obrazec projekta
Startup="Form1"                       ' zagonski obrazec
' preostale nastavitve projekta
```

Vse vključene zunanje datoteke so opremljene z oznako različice, kar omogoča enostavno nadgradnjo projekta na izpopolnjeno različico gradnika.

Omenimo še datoteke s podaljškom .VBW. V njih je kot navadno besedilo opisano število in razporeditev prikazanih oken projekta v razvojnem okolju. S to datoteko lahko visual basic obnovi videz razvojnega okolja natančno tako, kakor smo ga nazadnje zapustili.

Obrazec

Posamezen obrazec je shranjen v datoteki s podaljškom FRM. Ker gre za datoteko z besedilom, vanjo ni mogoče shraniti opisa binarnih sredstev, ki so vključena v obrazec. Tovrstna vsebina je zato shranjena v datoteki z enakim imenom, vendar s podaljškom .FRX. Če obrazec ne premore binarnih sredstev (npr. bitnih slik), datoteke FRX ni.

```
VERSION <različica>
Begin VB.Form <ime obrazca>
  ' lastnosti obrazca
  <lastnost> = <vrednost>
  ' preostale lastnosti obrazca, ki se razlikujejo od privzetih nastavitev

  ' predmet na obrazcu
  Begin VB.<razred> <ime predmeta>
      ' lastnosti predmeta, ki se razlikujejo od privzetih nastavitev
      ' lastnost, ki vsebuje binarno sredstvo
      <binarna lastnost> = "<obrazec>.frx":<odmik od začetka datoteke16>
  End

  ' preostali predmeti na obrazcu

End

' atributi obrazca
Attribute <ime atributa> = <vrednost atributa>

' programska koda obrazca
```

Modul s kodo

Moduli s kodo so shranjeni v datotekah s podaljškom .BAS. V glavnem je v njih programska koda, če pa modulu pripada še kak atribut, je ta določen na vrhu datoteke. Splošna oblika modula s kodo je:

```
' atributi modula
Attribute <ime atributa> = <vrednost atributa>

' programska koda modula
```

Razredni modul

Novi, lastni razredni moduli, ki jih razvijemo v visual basicu, so shranjeni v datoteki s podaljškom .CLS. Njena najpogostejša oblika se glasi:

```
VERSION <različica> CLASS
BEGIN
  ' lastnosti razrednega modula
  <lastnost> = <vrednost>
END
' atributi razrednega modula
Attribute <ime atributa> = <vrednost atributa>

' programska koda razrednega modula

' nekateri podprogrami imajo lahko dodatne atribute
Public Sub <ime podprograma>()
Attribute <ime atributa> = <vrednost atributa>
  ' koda podprograma
End Sub
```

Opis namestitve

Zadnja vrsta datotek, ki jo bomo opisali v tem dodatku, ne izvira iz razvoja projekta, ampak nastane kot posledica določenega poteka namestitve projekta, ki ga izdelamo z orodjem Application Setup Wizard. Potek namestitve je določen v datoteki SETUP.LST, katere najpogostejša oblika se glasi:

```
' opis namestitve datotek, potrebnih za nadaljevanje dela
[BootStrap]
File<n>=<disk>,<razcpljena>,<ime>,<pravo ime>,<ciljna mapa>,<registracija>,
<drugi atributi>

' opis namestitve datotek projekta
 [Files]
File<n>=<disk>,<razcpljena>,<ime>,<pravo ime>,<ciljna mapa>,<registracija>,
<drugi atributi>

' parametri namestitve
[Setup]
<lastnost> = <vrednost>
```

Stvarno kazalo

! 372
82, 86
\ 85
: 83
' 84
& 49, 76, 81, 87
* 86
.. 150
= 84
? 86
^ 85
_ 80, 83

A

AbsolutePosition (l) 365
abstrakcija podatkov 516
abstraktni razred 522
Access 350
AccessKeyPress (d) 598
AccessKeys (l) 598
Activate (d) 47
ActiveControl (l) 262
ActiveForm (l) 195, 262
ActiveX 11, 59, 541
  DLL 542
  EXE 553
  gradniki 186, 572
AddItem (m) 68
AddNew (m) 373
ADO 628
akcijske poizvedbe 447
Align (l) 185
Alignable (l) 598
Alignment (l) 51, 56
Alignment (l) 392
AllowAddNew (l) 391
AllowCustomize (l) 283
AllowDelete (l) 391
AllowSizing (l) 392
AllowUpdate (l) 392
AllowZeroLength (l) 407
AllPermisions (l) 429
ALTER TABLE 443
Ambient (r) 583
AmbientChanged (d) 583
And (vb) 86
animacija 270
  gradnik Animation (r) 270
App (r) 221, 222
AppActivate (vb) 155
Append (m) 402
AppendChunk (m) 419
Application (r) 629
Application Setup Wizard 339
Application Wizard 232
Application_OnEnd (d) 632
Application_OnStart (d) 632
Archive (l) 140
Arrange (m) 199
Array (vb) 105
ASCII 55
asinhrono
  povezovanje 480
  poizvedbe 481
  obveščanje 560
ASP 628
Assert (m) 168
atribut 141
Attributes (l) 404
AutoActivate (l) 383
AutoRedraw (l) 260
AutoShowChildren (l) 196
AutoSize (l) 51
AutoVerbMenu (l) 383
AVI 270
avtomatizacija (automation) 128, 379

Legenda

(vb) – element jezika visual basic (rezervirane besede, tipi, funkcije)
(sql) – element jezika SQL
(r) – razred/gradnik (class/control)
(l) – lastnost (property)
(z) – zbirka (collection)
(m) – metoda (method)
(d) – dogodek (event)

B

BackColor (l) 41, 248
BackStyle (l) 51, 248
barve 245
  paleta 245
  zapis 245
BeginTrans (m) 422
Beležka (Notepad) 53
beli prostor (whitespace) 231
binarna združljivost 569
binarne datoteke 313
BINARY 442
BIT 442
bitna operacija
  izrisovanja 248
  preračunavanje 87
BMP 289
BOF (l) 364
BOFAction (l) 358
Bookmark (l) 365, 390
Bookmarkable (l) 365
Boolean (vb) 97
BorderColor (l) 247
BorderStyle (l) 42, 61, 247
BorderWidth (l) 247
BoundColumn (l) 387
Button (r) 393
ByRef (vb) 111
BYTE (sql) 442
Byte (vb) 96
ByVal (vb) 111

C

CAB 608
  primarni 608
  sekundarni 608
Cancel (l) 56
Cancel (m) 480
CancelError (l) 192, 205
CancelUpdate (m) 373
CanGetFocus (l) 597
Caption (l) 43, 49
CBool (vb) 102
CByte (vb) 102
CCur (vb) 102
CDate (vb) 102
CDbl (vb) 102
CDec (vb) 102
celo število 81
celovitost podatkov 405, 443
Change (d) 52, 71, 142
CHAR (sql) 442
ChDir (vb) 140
ChDrive (vb) 140
Checked (l) 75
CInt (vb) 102
Circle (m) 252, 254
Class (l) 382
Clear (m) 68, 200, 215
ClearColumns (m) 393
Click (d) 62, 76
Clipboard (r) 200
ClipControls (l) 260
ClipMode (l) 386
CLng (vb) 102
Cls (m) 252
Col (l) 390
ColorMode (l) 264
Column (l) 390
Columns (l) 68
COM 624
Command (vb) 230
CommitTrans (m) 422
CompactDatabase (m) 412
Connect
  Connect (l) 356, 411
  niz 474
Connection (r) 479
CONSTRAINT (sql) 443
Container (l) 128
Containers (c) 428
Control (vb) 129
Controls (c) 127
Copies (l) 264
COUNTER (sql) 442
CREATE INDEX (sql) 445
CREATE TABLE (sql) 441
CreateDatabase (m) 402
CreateEmbedd (m) 385
CreateLinked (m) 385
CreateObject (vb) 616
CreateQueryDef (m) 463
CreateUser (m) 427
CSng (vb) 102
CStr (vb) 102
CUR 289
CurDir (vb) 140
CURRENCY (sql) 442
Currency (vb) 96
CurrentX (l) 244
CurrentY (l) 244
CVar (vb) 102
CVDate (vb) 102
CVErr (vb) 102

Č

čakajoča zahteva 619
čarovnik (Application Setup Wizard) 340
član (member) 497
  opis 517
  privzeti 516
členjenje (parsing) 230
črta (Line) (r) 247

D

DAO 397
Data (r) 354
Database (r) 402
DatabaseName (l) 356
DataChanged (l) 361
DataField (l) 361
DataMode (l) 394
DataSource (l) 361
DataUpdatable (l) 372
Date (vb) 97

DATETIME (sql) 442
datotečni strežnik 467
datotečni sistem 138
datoteka
  branje 206
  CAB 347
  Global.ASA 632
  podaljšek 192
  podpis 311
  sredstev (resource file) 287
datoteke
  binarne 206, 313
  deljenje (sharing) 207
  shranjevanje 208
  vrste 206
  zaporedne 207
datumska vrednost 82
datumski niz 82
DBEngine (r) 377
DBGrid (r) 390
DblClick (d) 47
DCOM 572
Deactivate (d) 47
Debug (r) 168
Declare (vb) 225
dedovanje (inheritance) 495, 521
Default (l) 56
DefaultCursorDriver (l) 488
DefaultCursorType (l) 360
DefaultExt (l) 192
DefaultPassword (l) 479
DefaultType (l) 360, 479
DefaultUser (l) 479
DefaultValue (l) 392, 407
DefColWidth (l) 392
definicija podatkov 399
dejavna pot 138
dejavne strežene strani 628
DELETE (sql) 453
Delete (m) 377, 409
DeleteSetting (l) 159
delovna površina (client area) 198
delovna skupina (workgroup) 425
delovni prostor (workspace) 402, 423
  neposreden 479
Description (l) 214
Developer Studio 8
DeviceName (l) 264
DialogTitle (l) 192
Dictionary (c) 626
Dir (vb) 145
DisabledPicture (l) 58
DisplayType (l) 383
dnevnik (log) 222, 223
Do (vb) 121
Documents (c) 428
dodajanje zapisov 373
DoEvents (vb) 325
dogodek (event) 37, 39
  lastni 518
  proženje lastnih 518
  uporabniški 37
  vrstni red 47
dogodkovna naravnanost 38
domorodne zbirke 351
doseg (scope) 91
  globalni 91
  modulski 91
  podprogramski 91
dostopna tipka (access key) 49
  določanje 598
dostopno zaporedje (tab order) 50
DOUBLE (sql) 442
Double (vb) 96
DoVerbs (m) 383
dovoljenje (license key) 603
DownPicture (l) 58
DrawMode (l) 248, 251
DrawStyle (l) 251
DrawWidth (l) 251
Drive (l) 138
DROP TABLE (sql) 443
drsnik (ScrollBar) (r) 70
Duplex (l) 264
Dynamic 414
Dynaset (r) 413

E

Edit (m) 373
EditMode (l) 374
eksplicitna najava 90
Empty (vb) 99
Enabled (l) 45, 72
End (vb) 132
EndDoc (m) 266
EnterFocus (d) 597
Enum (vb) 98
EOF (l) 365
EOFAction (l) 358
Eqv (vb) 87
Err (r) 214
Error (d) 363
Error (vb) 99
Errors (r) 377
Event (vb) 518
Exclusive (l) 358
EXEName (l) 223
Exit Do (vb) 123
Exit For (vb) 123
Exit Function (vb) 110
Exit Sub (vb) 109
ExitFocus (d) 597
Extender (r) 583

F

faza razhroščevanja 160
FetchVerbs (m) 384
Field (r) 371, 403
Fields (c) 371
FieldSize (l) 419
FileCopy (vb) 163
FileDateTime (vb) 163
FileLen (vb) 163
FileName (l) 194
FileSystemObject 626
FileTitle (l) 194
FillColor (l) 249
FillStyle (l) 249
Filter (l) 193, 417

FilterIndex (l) 193
FindFirst (m) 369
FindLast (m) 369
FindNext (m) 369
FindPrevious (m) 369
Flags (l) 193
Font (l) 51
FontCount (l) 262, 264
Fonts (l) 262, 264
For Each (vb) 122
For (vb) 122
ForeColor (l) 251
Form Designer 9, 19
Form (vb) 129
Form Layout 13
Format (vb) 175
Format (l) 392
Forms (c) 127
Forward-only 414
ForwardFocus (l) 597
FreeFile (vb) 207
Friend (vb) 565
Function (vb) 109
funkcija 108

G

geslo 426
  zbirke podatkov 430
Get (vb) 315
GetAllSettings, p 159
GetAttr (vb) 147
GetChunk (m) 419
GetData (m) 200
GetFormat (m) 200, 201
GetObject (vb) 616
GetRows (m) 417
GetSetting (vb) 158
GetText (m) 200
glavno okno 195
Global.ASA 632
GotFocus (d) 54
graditelj razredov 538
gradnik (control) 10, 541
  ActiveX 572
  dodatni 186
  grafični 247, 250
  omogočenost 45
  polje 131
  poravnava 598
  preurejanje 21
  prosojnost 598
  risanje 20
  temeljni 581
  uporaba 41
  vidnost 45
  zunanji 186
grafični uporabniški vmesnik (GUI) 37, 240
GUID 420, 442, 569
gumb (CommandButton) (r) 20, 55

H

hDC (l) 272
Height (l) 262
Help Workshop 331
HelpContext (l) 214
HelpContextID (l) 335
HelpFile (l) 214, 334
Hidden (l) 140
Hide (m) 45, 178
HoldFields (m) 393
hrošč (bug) 159
HTML 628
HTTP 629
hWnd (l) 227

I

IDE (združeno razvojno okolje)
  elementi vmesnika 8
  okno s kodo 24, 33
  grafični 250
  poravnava gradnikov 30
  preurejanje oken 16
  preurejanje vrstic 14
  prilagajanje 14
  prilagoditev okna s kodo 16
  projektno okno 11
  stanje 9
  zaklenjeni položaj 30
If (vb) 119
ikona aplikacije 218
ImageList (r) 271
Imenik 138
Immediate Window 168
Imp (vb) 87
Implements (vb) 522
indeks 404
Index (l) 59
Index (r) 404
Initalize (d) 197
InitDir (l) 193
Initialize (d) 46
Initialize (d) 502
InitProperties (d) 578
Input # (vb) 207
Input (vb) 207
InputBox (d) 171
INSERT INTO (sql) 455
Instancing, p 543
InStr (vb) 203
INT (sql) 442
Integer (vb) 96
IntegralHeight (l) 64
Internet Information Server 628
Interval (l) 72
InvisibleAtRuntime (l) 600
Is (vb) 133
IsArray (vb) 105
IsDate (vb) 101
IsEmpty (vb) 101
IsError (vb) 101
iskanje
  hitro 370
  zaporedno 368
IsMissing (vb) 115
IsNull (vb) 101
IsNumeric (vb) 101
ItemCheck (d) 69

ItemData (l) 63
izbirne poizvedbe 447
izhodiščni naslov (base address) 570
izpeljava vmesnika 522
izpis
  oblikovan 175
izraz (expression) 84
  predmetni 133
  primerjalni 87
  prireditveni 84
izrezovanje (clipping) 260
izrisovanje (painting) 250
  besedila 255
izvajalna napaka (run time error) 212
izvajalne zahteve 340
izvod (instance) 125
izvršilno okno (immediate window) 16

J

javna poizvedba 465
JET 350
Jet SQL 460
JScript 624

K

kaskadno
  dopolnjevanje 406
  odstranjevanje 406
kazalec (cursor) 352, 487
  miške, uporabniški 43
KeyCode 154
KeyPress (d) 54
KeyPreview (l) 45, 153
Kill (vb) 163
KillDoc (m) 266
klicni sklad 215
ključ (key) 400
  primarni 400
  tuj 400
koš (recycle bin) 168
koda (code) 24, 31
kodni modul (code module) 93
  dodajanje 93
  odstranitev 93
kolizija (collision) 484
kombiniran seznam 65
komentar 83
konstanta (constant) 89
  najava 92
  začetna vrednost 92
  SQL 436
konstantna vrednost (literal) 81
koordinatni sistem 240
koračno izvajanje 161, 510
kriptiranje 403

L

LargeChange (l) 70
LastDLLError (l) 214
LastModified (l) 374
lastnik 429
lastnost (property) 13, 39
  nastavljanje 22
  podprogramska 508
  privzeta 372
  strani 585
Left (vb) 148
lik (Shape) (r) 247
Like (vb) 85, 436
Line Input # (vb) 207
Line (m) 252, 253
List (l) 63
ListCount (l) 66
ListField (l) 387
ListIndex (l) 66, 141
ListView (r) 275
Load (d) 32, 46
Load (vb) 131
LoadPicture (vb) 221, 246
LoadResData (vb) 291
LoadResPicture (vb) 282
LoadResString (vb) 291
ločilo 83
Locked (l) 392
LockEdits (l) 375
LOF (vb) 207
LogEvent (m) 224
logična vrednost 82
LogMode (l) 223
LogPath (l) 223
LONG (sql) 442
Long (vb) 96
LONGBINARY (sql) 442
LongBinary 419
LONGTEXT (sql) 442
LostFocus (d) 54

M

mapa 138
Mask (l) 385
MaskColor (l) 58
Max (l) 70
MaxLength,l 53
MDI 8, 195
  vmesnik 8, 181
MDIChild (l) 185
MDIForm (vb) 129
Me (vb) 134
medpomnjenje (caching) 486
Memo 419
meni (Menu) 73
  izbira 73
  načrtovanje 73
  poimenovanje 74
  urejevalnik 73
metoda (method) 39
  grafične 250
  lastna 514
  privzeta 534
Microsoft Excel 177
Mid (vb) 149
miška
  nizki nivo 299
Min (l) 70
MkDir (vb) 163

Mod (vb) 85
modalnost 174
modul 91
MousePointer (l) 43
Move (m) 51, 367
Moveable (l) 44
MoveFirst (m) 367
MoveLast (m) 367
MoveNext, m 367
MovePrevious (m) 367
mreža (grid) 31
MSFlexGrid (r) 390
MsgBox (d) 170
MultiLine (l) 53
MultiSelect (l) 67

N

načrtovalec (designer) 466
načrtovanje 12
  relacijske zbirke 399
nadzorni meni (system menu) 12
Name (vb) 163
namestitev (setup) 5, 338
  čarovnik 340
  orodja 346
  potek 346
  predloga 345
namig (tooltip) 51
namizje (desktop) 138
namizna zbirka 467
napake
  prestrezanje pri avtomatizaciji 617
  prestreznik 212
  proženje 214
napis (label) (r) 31
naravni vrstni red 353
naštevni
  predmet (enumarator) 536
  tip (enumeration) 98
navidezni pomnilnik (virtual memory) 570
neposreden delovni prostor 479
neresnica (false) (vb) 82
neveljaven zapis 368
nevezan način 394
New (vb) 130
NewIndex (l) 68
NewPage (m) 266
niz (string) (vb) 83, 97
  določene dolžine 97
  preiskovanje 203
  spremenljive dolžine 97
NoMatch (l) 369
NonModalAllowed (l) 553
Normal (l) 140
normalizacija 399
Norton Commander 143
Not (vb) 86
Nothing (vb) 130
Null (vb) 99, 100
Number (l) 214
NumberFormat (l) 392

O

Object (vb) 129
ObjectAcceptFormats (m) 384
ObjectGetFormatsCount (m) 384
ObjectVerbsCount (m) 384
oblikovan izpis 175
oblikovano
  besedilo 276
  vnosno polje (MaskedEdit) (r) 385
obnavljanje risbe 304
obrazec (form) 19, 41
  dodajanje 172
  izvod 196
  podrejen 184
  skrivanje 178
  življenski krog 504
obsežna vrednost (large value) 419
obsežna vsebina 378
obstojna grafika 258
OCX 187, 541
ODBC
  gonilniki 470
  nadzor 470
  protokol 352
  vzpostavitev vira 473
  zbirke 352
oddaljena avtomatizacija 572
odjemalec 467, 541
odločitveni stavek 119
odložišče (Clipboard) 59, 200, 384
  prenos grafike 317
odprti seznam (ListBox) (r) 63
odstranjevanje zapisov 377
odvisnost (dependency) 340
Okenski komandant 137
Okna 37
Okna 95, vmesnik 267
okno (window) 37
okvir (Frame) (r) 56
OLE 541
OLE (r) 378
OLE_CANCELBOOL (vb) 591
OLE_COLOR (vb) 591
OLE_OPTEXCLUSIVE (vb) 591
OLE_TRISTATE (vb) 591
OLEDropAllowed (l) 382
OLERequestPendingMsgText (l) 620
OLERequestPendingMsgTitle (l) 620
OLERequestPendingTimeout (l) 620
OLEServerBusyMsgText (l) 620
OLEServerBusyMsgTitle (l) 620
OLEServerBusyRaiseError (l) 620
OLEServerBusyTimeout (l) 620
OLETypeAllowed (l) 383
On Error Goto (vb) 212
On Error Resume Next (vb) 213
opcijsko stikalo (OptionButton) (r) 56
Open (vb) 207
OpenConnection (m) 480

OpenDatabase (m) 409
OpenRecordset (m) 414
operacijski sistem 37
operator 84
  logični 86
  primerjalni 85
  prioriteta 87
  računski 84
  SQL 436
opis člana 517
Option Base (vb) 104
Option Compare (vb) 86
Option Explicit (vb) 90
Optional (vb) 116
Options (l) 359
Or (vb) 87
Oracle 470
Orientation (l) 264
OriginalValue (l) 484
orodjarna (toolbox) 10
osmiški zapis 81
Owner (l) 430
označevalec (marquee) 29, 300
oznaka
  gradnik (label) (r) 49
  razreda (dispatch ID) 529
  zbirke zapisov (bookmark) (l) 365, 374
  vrstice 212

P

Page (l) 264
Paint (d) 47
Paint (d) 260
PaintPicture (m) 257
paketno posodabljanje (batch updating) 483
paleta barv 16
Palette (l) 246
PaletteMode (l) 246
PaperBin (l) 264
PaperSize (l) 264
ParamArray (vb) 117
parameter 111
  nedoločljivo število 117
  neobvezni 115
  posredovanje 111
  seznam 25
PARAMETERS (sql) 464
parametrizirane poizvedbe 463
PasswordChar (l) 53
PasteSpecialDlg (m) 385
Path (l) 139, 223
PathChange (d) 142
Pattern (l) 140
PatternChange (d) 142
PercentPosition (l) 365
Permissions (l) 429
Picture (l) 44, 58, 221, 249
PID 426
piksel 242
piškotek (cookie) 629
podatkovna zbirka, *glej* zbirka podatkov
podatkovni strežnik 467
podatkovni stroj (database engine) 351
podatkovni tip 95
  pretvorba 102
  prilagodljiv 99
  sestavljeni 103
  številski 95
  uporabniško določen 107
podprogram 108
  klic 113
  odzivni 25
  splošni 25, 108
podprogramska lastnost 508
podrejeno okno (child window) 12, 184
pogon (drive) 138
Point (m) 252
poizvedba (query) 353
  izgradnja 466
polje (array) 103, 403
  dinamično 106
  gradnikov (control array) 60, 131
  statično 103
  večrazsežno 105
  vsebovano 104
pomnilniška stran 571
pomoč
  pojasnjevanje 335
  prevajanje 331
  priprava besedila 328
  razvoj 327
  sprotna 40, 327
popolno prevajanje 130, 556
popravljanje zapisov 373
popravljanje zbirke 412
PopUpMenu (m) 76, 308
Port (l) 264
poselitev zbirke 367
pot (path) 139
povezani predmet 378
povezava 474
povezovanje (binding) 528
  pozno 528
  zgodnje 528
povratni klic (callback) 567
pozdravno okno (splash screen) 324
pravice 428
pravila vnosa 407
predkupna večopravilnost 542
predloga (template) 28, 189
  mape 190
  namestitve 345
  pripravljanje 190
predmet (object) 125, 496
  izraz 133
  notranji 527
  splošna spremenljivka 129
  spremenljivka 128
  stanje 126
  uničenje 133
predmetna analiza 500
predmetni brskalnik 497
predmetni model 529

DAO 397
predpone poimenovanja 182
preglednica
nevezana 394
obnovitev vsebine 393
urejanje 391
vezana 390
preiskovanje
hitro 370
zaporedno 353
prekinitvena točka (breakpoint) 159
prekoračitev (overflow) 302
prelivanje zapisov (data pipelining) 485
premiki 367
prepis slike 258
prepisovanje zapisov 417
prepuščena poizvedba 476
Preserve (vb) 106
prestavljanje (rebasing) 570
prestrezanje napak
način 511
prevajanje 217
domorodna koda 219
preverjanje
najav 89
vnosa 408
PrevInstance (l) 223
prijateljski podprogram (friend subroutine) 507, 565
prijava 423
priključena tabela 410
Primarni ključ 400
Print # (vb) 210
Print (m) 168, 252, 255
Printer (r) 251, 263
Printers (c) 127, 263
PrintForm (m) 46
PrintQuality (l) 265
prioriteta operatorjev 87
priročni meni (context menu) 9, 308
privilegirani uporabnik 427
privzeto
član 516
lastnost 372
vmesnik 525
procedura 108
proces 542
programska koda 79
ProgressBar (r) 268
projekt 11
PromptChar (l) 386
PromptInclude (l) 386
Property (vb) 509
PropertyBag (r) 584
PropertyChanged (m) 582
prva aplikacija 19
PSet (m) 252
Put (vb) 313

Q

QueryDefs (c) 463
QueryUnload (d) 47

R

RAD 541
Raise (m) 215
RaiseEvent (vb) 519
Rastriranje (dithering) 245
razcep (split) 393
razdrobljenost 411
razhroščevalnik 159, 510
Raziskovalec (Explorer)
vmesnik 183
različice projekta 569
razred (class) 125
abstraktni 522
izvod 496
obnašanje 126
privzeti član 516
ustvarjanje 495
razredni modul 499
razveljavitev (undo) 316
RC 289
RDO 469
ReadFromFile (m) 385
ReadOnly (l) 140, 358
ReadProperties (d) 578
realno število 81, 96
ReBind (m) 393
RecordCount (l) 365
Recordset (l) 360
Recordset (r) 413
uporaba 364
RecordsetType (l) 359
RecordSource (l) 356
ReDim (vb) 106
referenca 496
referenčna celovitost 405
Refresh (m) 52, 299, 361, 409
RegisterDatabase (m) 473
rekurzija 112
relacija 400, 406
relacijska poizvedba 433
Relation (r) 406
Rem (vb) 84
RemoveItem (m) 68
RepairDatabase (m) 412
Reposition (d) 364
Requery (m) 485
Request (r) 629
Required (l) 407
Resize (d) 47
resnica (true) (vb) 82
Resource Compiler 287
Response (r) 629
Resume Next (vb) 213
Resume (vb) 213
ReturnsRecords (l) 477
rezervirana beseda (keyword) 80
rezervna kopija 411
RGB (vb) 245
RichTextBox (r) 276
risanje 299
vmesnika 19
Risar 278
RLE 270
RmDir (vb) 163

ročica (handle) 21
risalne površine 272
RollBack (m) 422
Row (l) 390
RowBuffer (r) 395
RowHeight (l) 392
RowSource (l) 387
RTF 277

S

SavePicture (vb) 322
SaveSetting (vb) 158
SaveToFile (m) 385
Scale (m) 242
ScaleHeight (l) 243
ScaleLeft (l) 243
ScaleMode (l) 241
ScaleTop (l) 243
ScaleWidth (l) 243
ScaleX (m) 244
ScaleY (m) 244
Screen (r) 262
Scroll (d) 71
ScrollBars (l) 54
SDI, vmesnik z enim dokumentom 8, 183
Seek (m) 370
SelBookmarks (c) 394
SELECT (sql) 448
Select Case (vb) 120
SELECT INTO (sql) 453
Selected (l) 67
SelectionChanged (d) 586
SelEndCol (l) 394
SelLength (l) 201
SelStart (l) 201
SelStartCol (l) 394
SelText (l) 200
SendKeys (vb) 155
Server (r) 629
Session (r) 629
Session_OnEnd (d) 632
Session_OnStart (d) 632
sestavljene poizvedbe 455
Set (vb) 130
SetData (m) 200
SetFocus (m) 46, 54
SetText (m) 200
Setup.LST 347
Setup1.EXE 347
seznam map (DirListBox) (r) 139
seznam
vezan 386
znakov 86
seznam pogonov (DriveListBox) (r) 138
Shape (r) 247
Shift 154
SHORT (sql) 442
Shortcut (l) 76
Show (m) 45, 174
ShowColor (m) 194
ShowFont (m) 194
ShowHelp (m) 194, 337
ShowOpen (m) 194
ShowPrinter (m) 194
ShowSave (m) 194
shranjevanje 27
samodejno 28
SID 426
sidranje (docking) 16
simbolično ime (identifier) 80
pravilno 80
SINGLE (sql) 442
Single (vb) 96
sinonim tabele 457
sistemska administracija 626
sistemska paleta 245
sistemska podatkovna zbirka 425
SizeMode, p 383
skala (scale) 241
sklicevanje 117
posredno 118
skrivanje podrobnosti (encapsulation) 507
skupina uporabnikov 426
Slider (r) 268
slika (Image) (r) 221
sloji obrazca 259
SmallChange (l) 70
Snapshot 414
Sort (l) 417
Sorted (l) 64
Source (l) 214
SourceDoc (l) 383
SourceItem (l) 383
SourceTableName (l) 411
splet (WWW) 607
sporočilo (message) 37
sporočilna zanka 553
spremenjivka (variable) 88
dinamična 92
doseg 91
globalni doseg 94
najava 89, 92
predmetna 128
privzeta vrednost 89
statična 92
življenjska doba 91
SQL 433
indeks 444
izrazi 436
konstante 436
operatorji 436
relacije 445
stavek 434
tabela 441
ukaz 434
zunanja tabela 453
SQL Server 470
sredstvo (resource) 287
dodajanje v projekt 290
nizi 291
uporaba 290
SSTab (r) 274
standardna pogovorna okna 191
StartLogging (m) 224
StartMode (l) 566
StartUpPosition (l) 44
statična lastnost 507
statična spremenljivka 92

stavek (statement) 79
  odločitveni 119
  zančni 122
stikalo (CheckBox) (r) 56
StillExecuting (l) 480
stolpec (column) 352, 443
Stop (vb) 160
stran (page) 375
stran lastnosti (property page) 585
strežnik 541
  oddaljeni 572
  samostojna aplikacija 566
  večnitnost 571
  znotraj procesni 542
  zunaj procesni 553
Stretch (l) 250
String (vb) 97
struktura 107
Style (l) 57, 64
Sub (vb) 108
System (l) 140
System.MDB 425
SystemDB (l) 426

Š

šestnajstiški zapis 81
šifrant (look-up table) 387
števec 122
številka različice 218

T

TabAction, p 392
Tabela 352
Table (r) 413
TableDef (r) 402
TabStrip (r) 274
Tag (l) 61
temeljni gradnik 581
Terminate (d) 49, 197
TEXT (sql) 442
Text API Viewer 226
Text (l) 53, 65
TextHeight (m) 257
TextStream (r) 626
TextWidth (m) 257
Timer (d) 72
Timer (vb) 374, 481
tiskalnik 263
Title (l) 222
tok (stream) 626
tok izvajanja 118
ToolboxBitmap (l) 602
ToolTipText (l) 51
TrackDefault (l) 265
Transactions (l) 422
transakcija 421
TreeView (r) 274
trenutni zapis 352
tuj ključ 400
twip 242
TwipsPerPixelX (l) 262
TwipsPerPixelY (l) 262
Type (vb) 107
TypeOf (vb) 134

U

UCase (vb) 148
ukazna vrstica 230
  parametri 229
UnboundAddData (d) 395
UnboundDeleteRow (d) 395
UnboundGetRelativeBookmark (d) 395
UnboundReadData (d) 394
UnboundWriteData (d) 395
UNC 139
Unload (d) 47
Unload (vb) 131
Until (vb) 121
Updatable, p 372, 409
UPDATE (sql) 454
Update (m) 373
UpdateControls (m) 361
UpdateRecord (m) 361
UpDown (r) 269
uporabnik 426
uporabniško ime 426
upravljanje datotek 163
upravnik opravil (Task Manager) 48
ura (Timer) (r) 72
urejanje zapisov 417
Urejevalnik 181
UseMaskColor (l) 58
UserMode (l) 584
UserName (l) 429
Users (c) 427
Usodna napaka (fatal error) 212
UUID 569
uveljavitev (redo) 316
uvodno okno 7

V

Validate (d) 362
ValidateOnSet (l) 407
ValidationError (l) 386
ValidationRule (l) 407
ValidationText (l) 407
Value (l) 57, 70, 393, 484
VARCHAR (sql) 442
Variant (vb) 99
  napaka 100
  nedoločena vrednost 100
  opredelitev tipa 101
  polje 104
  prazna vrednost 99
Varnost.MDW 425
varnostni mehanizem 425
VarType (vb) 101
VBA 612
VBS 612
VBX 541
večličnost (polymorphism) 202, 521
večnitnost 571

večopravilnost (multitasking) 38
veljaven indeks 416
vezan seznam 386
vezani gradniki 354
vgrajene funkcije 437
vhodni vmesnik 518
virtualna tabela (vtable) 529
Visible (l) 45
VisibleValue, p 484
visual basic 4
  popravki (Service Pack) 6
  različice 5
Visual Basic for Applications 623
Visual Basic Scripting 624
Visual Data Manager 401
vizitka (about) 217
vmesna koda 219
vmesna kopija (copy buffer) 373
vmesnik 496, 500
  privzeti 525
  več vmesnikov 525
vnosna točka (focus) 34, 50
  obravnava 597
vnosno polje (textbox) (r) 29, 53
vrstica (row) 352, 443
vrstica gumbov (toolbar) 9
  preurejanje 10
vsebnik (container) 41, 61, 128
  grafični 251
  OLE 378
vsebovani predmet 378
vstavljeni predmet (insertable object) 378
vstavljeno popravljanje (in-place editing) 378
vzorec (pattern) 86, 140
vzpostavitev (instantiation) 46

W

WhatsThisButton (l) 336
WhatsThisHelp (l) 336
WhatsThisHelpID (l) 336
WhatsThisMode (l) 336
While (vb) 121
Width (l) 262
WindowList (l) 76, 189
Windows 37
Windows API 225
Windows Scripting Host 626
WindowState (l) 44
With (vb) 118
WithEvents (vb) 519
WordWrap (l) 51
Workspace (r) 423
WrapText (l) 392
Write # (vb) 210
WriteProperties (d) 578

X

X1 (l) 247
X2 (l) 247
Xor (vb) 87

Y

Y1 (l) 247
Y2 (l) 247

Z

zadolžitev (delegation) 527, 533
zagon 26
zaklepanje
  optimistično 376
  parameter 415
  pesimistično 375
  zapisov 375
  zbirke 430
zanka
  prekinitev 123
  števec 122
zapis 352
  nabor vrednosti 407
zaposlen gradnik 619
zaprti seznam (ComboBox) (r) 63
zaslon 262
zaustavitev 27
zbirka (collection) 122, 126, 529
  lastna 532
  zapisov (recordset) 353, 413
zbirka podatkov (database) 350
  popravljanje 412
  struktura 433
  zgoščevanje 412
  zgradba 352
združeno razvojno okolje, IDE 7
združevalne funkcije 437
združevanje
  nizov 87
  poizvedb 460
  tabel 457
zgoščevanje 412
znak 83
Zoom (l) 265
ZOrder (m) 52
zunanja
  tabela 410
  zbirke 352

Ž

življenjska doba (lifetime) 91

Visual basic, sodobno programiranje, ponatis v elektronski obliki
Avtor: Uroš Mesojedec
Lektorirala: Dora Mali
Uredil: Matjaž Potrč
Založila: Založba Pasadena, d. o. o., Dolenjska cesta 244, Ljubljana,
telefon (01) 280 53 50, www.pasadena.si, info@pasadena.si
Oblikovanje: Peter Gedei
Ljubljana, maj 2002