Delphi 6 Vodič za programere

Savladajte objektno orijentisani Pascal

Kamen temeljac kvalitetnog programiranja jesu dobre alatke. Object Pascal je jedna od njih. Kao nastavak štiva iz prethodnog poglavlja ili onoga što već znate, u ovom poglavlju ćemo ilustrovati veštine objektno orijentisanog programiranja koje su neophodne za svaki program. Delphi predstavlja izuzetnu alatku zasnovanu na Object Pascalu. U ovom poglavlju naučićete kôd koji je osnov svakog programa.

Konvencije usvojene u Delphiju

Pre četvrt veka jezik C je bio udarna alatka. U to doba prevodioci su, kao i onaj u jeziku C, bili neznatno tipizirani. U kodu ste mogli da deklarišete promenljive kao pokazivače i da ih prosle?ujete argumentima očekujući cele brojeve, ili obrnuto. Na primer, celobrojna promenljiva kojoj je dodeljena vrednost nula mogla je slučajno biti dodeljena tipu *char** (u žargonu jezika C, pokazivaču na znak). To se doga?alo zato što prevodioci nisu podržavali striktnu upotrebu tipova, a to ne čine ni danas. Na kraju krajeva, podaci su samo brojevi. Problem je u tome što se, kada pristupite celom broju kao pokazivaču, na?ete na samom početku memorije rezervisane za BIOS (osnovni ulazno-izlazni sistem), što je veoma rizično. Drugi neugodan problem su globalne promenljive. Većina programera (ako to nisu zapisali) ne mogu da se sete tipa podatka ili promenljive deklarisanih mesec dana ranije u kodu nekog drugog programera.

Nekoliko godina kasnije, nakon miliona dolara potrošenih na otklanjanje grešaka prouzrokovanih globalnim promenljivama i zloupotrebljenim celim brojevima i pokazivačima, pojavilo se rešenje. Početkom osamdesetih godina Microsoft je zaposlio Charlesa Simonyija iz Xerox Parca. Ma?ar Simonyi je smislio i popularisao takozvanu Ma?arsku konvenciju za dodeljivanje imena koja je, uspostavljajući sistem prefiksa, omogućila da se identifikuju tipovi podataka promenljivih. Smatralo se da programer, kada vidi prefiks, neće pogrešno upotrebiti pokazivač ili ceo broj. Na primer, tip *char** upotrebljen za čuvanje znakovnog niza koji se završava nulom, mogao bi dobiti prefiks *sz* . Nadobudni programer treba samo da zapamti da *sz* znači znakovni niz koji se završava nulom (engl. s-tring z-ero). I globalna promenljiva ima prefiks, pa ako možete da zapamtite šta znači *lpsz* , onda nećete morati da tražite deklaraciju promenljive da biste utvrdili kog je tipa.

Naznake i nenametljiva uputstva dobra su stvar, ali se i sami zapitajte da li svi poštuju crveno svetlo na semaforu, da li ulaze na prednja vrata tramvaja ili iskorišćenu tramvajsku kartu bacaju u korpu za otpatke. Verovatno i sami kršite neko slično nepisano ili nenametljivo napisano pravilo. No, bez obzira na rečeno, konvencija o prefiksima bila je najbolje što se u to vreme moglo postići na industrijskom nivou, tako da je prihvaćena i otada živi sopstvenim životom.

U to vreme se nije vodilo računa (a šta bi, inače, u tehnologiji opstalo dve decenije?) o činjenici da je danas većina prevodilaca strogo tipizirana. To znači da ne možete slučajno da previdite moguću nekompatibilnost tipova podataka; prevodilac vam to neće dopustiti. Isto tako, globalne promenljive se više ne

koriste tako često kao nekada. Ako očekuje ceo broj, znak ili znakovni niz, prevodilac će insistirati da ga dobije. Me?utim, izgleda da neki proizvo?ači softverskih alatki nikada ne odustaju. U Windowsovom API-ju, Visual Basicu i verovatno na još mnogo mesta, sve je preplavljeno prefiksima. Ima ih tako mnogo da je teško verovati da od njih neko može da napravi i održava jedinstven standard. Srećom, kod Inprisea nema neprijatnih, komplikovanih ili nerazumljivih prefiksa. Delphijev prevodilac je sjajan; pratite strategiju koju izlažemo u ovom poglavlju i prefiksi vam neće ni trebati.

Manje znači više

Delphi ne nameće konvenciju o imenovanju prefiksa već je koristi za objašnjavanje namene korišćenja podataka, umesto za deklarisanje tipova. Pravila dodeljivanja imena su korisna kada postoje ograničenja dužine imena promenljivih, kada strukturno programiranje nameće upotrebu globalnih promenljivih, a prevodilac ne može da iza?e na kraj sa pogrešnim korišćenjem podataka. No, već mnogo godina ništa od pomenutog ne stvara probleme. Objekti omogućavaju da izbegnemo korišćenje globalnih promenljivih, ograničenje dužine imena promenljivih ne postoji, a prevodioci ne dozvoljavaju pogrešnu upotrebu podataka. Zbog toga se u Delphiju komplikovana pravila dodeljivanja prefiksa izbegavaju kao izlišna. Umesto da pokušavate da ih sve držite u glavi, možete da se usredsredite na programiranje.

Najbolje tehnike

Jedna izreka kaže: "Kada čoveku daš ribu, nahranio si ga danas; ako ga naučiš da lovi ribu, nahraniće ga za ceo život". Isto važi i ovde. Naučite dvadesetak prefiksa i čim izmisle nove, moraćete da naučite još dvadesetak; stvarno ćete biti "siti" onoliko dugo koliko "živi" jedna grupa prefiksa.

Me?utim, ako se držite nekoliko zdravorazumskih pravila koja se lako pamte i lako primenjuju, to će vam pomoći da pišete čist, dosledan kôd.

1. Imenujte procedure pomoću imenica i glagola. Glagol opisuje akciju, a imenica opisuje podatke sa kojima se radi. Ukoliko upotreba glagola ne dolazi u obzir, onda se verovatno radi o podacima.
2. Obezbedite da procedura radi samo jedan posao, ali dobro; naziv tog posla treba da stavite u ime procedure.
3. Neka metode budu kratke i neće biti mogućnosti da pomešate promenljive.
4. Kada su metode kratke, manja je mogućnost da do greške do?e na nivou procedure.
5. Ne koristite globalne promenljive.
6. Izbegavajte skraćenice - upotrebljavajte cele reči.

To je to. Potrebno je malo vežbe da bi se pisale kratke funkcije, ali ćete, ako to uradite, manje vremena trošiti na otklanjanje grešaka. Korišćenje celih reči, imenica i glagola samo znači da treba da se opredelite za čitljivost. *IzracunajPorez* , *OtvoriBazuPodataka* ili *OcitajSvojstvo* jesu razumljive deklarativne naredbe. Zaista, potrebno je nešto vežbe, ali verujte da je ovo najbolji način koji će se kasnije isplatiti.

Konvencije

Iako ne postoje tajanstveni prefiksi, kao što je *lpsz* , programeri Inprisea se pridržavaju nekoliko pravila koja ćete i vi smatrati korisnim. Na primer, kada je nešto tip, dobija prefiks *T* , npr. *TForm* . Kada uklonite *T* , dobićete *Form* , što je zgodno ime za promenljivu, a i odmah vam kaže klasu objekta. Za drugi primer uzećemo polja. Polja u ovom kontekstu označavaju privatne podatke. (U odeljku "Specifikatori pristupa" saznaćete više o privatnim podacima.) Imenima polja se dodaje prefiks *F* ; ispustite ga i dobićete zgodno ime za svojstvo. (O svojstvima govorimo u odeljku "Dodavanje svojstava klasama".)

Poslednje pravilo o kome ćemo govoriti nalaže da članovi nabrojanog skupa dobijaju za prefiks prvi od dva inicijala u imenu skupa (Delphijevi nabrojani skupovi obično imaju ime sastavljeno od dve reči sa početnim velikim slovima). Evo jednog primera iz klase Form:

```
type TFormStyle = (fsNormal, fsMDIChild, fsMDIForm, fsStayOnTop);
```

FormStyle je svojstvo objekata klase TForm, odnosno obrazaca. Obratite pažnju na prefiks *T* . Ako poznajete ime tipa, lako ćete saznati i ime instance. Instanca promenljive tipa *TFormStyle* biće *FormStyle* . Primećujete li da svi članovi nabrojanog skupa imaju prefiks *fs* ?

Ovim završavamo sa pravilima. Podsetimo se, Delphi ih ima tri: prefiks za polja, tj. za privatne podatke je *F* , prefiks tipova je *T* , a prefiks članova nabrojanog skupa čine inicijali koji predstavljaju njihov važeći tip. Ova pravila se preporučuju, ali nisu obavezujuća. Konvencije su korisne samo ako vam život čine lakšim, ako se lako pamte, ako su bezmalo univerzalno primenljive i ako poboljšavaju vaš proizvod.

Sastojci svakog Windowsovog programa

Svaki program namenjen Windowsu mora da se uklopi u način rada ovog operativnog sistema. Zbog toga svaki program mora da ima neke osnovne delove. Windows je operativni sistem koji reaguje na doga?aje i šalje poruke. Osim što sadrže procedure, funkcije i podatke, svi programi za Windows moraju na neki način da odgovore na poruke koje šalje ili prima Windows kao reakciju na neki unutrašnji ili spoljni doga?aj.

Svi programi za Windows obično sadrže još nešto zajedničko. Kao što sledi iz imena ovog operativnog sistema, programu je potreban prozor, tj. grafičko okruženje. Obe ove stvari imaćemo na umu dok vam budemo pokazivali kako programiranje u Delphiju može da bude zabavno.

Grafičko okruženje

Proizvod koji je inače pristojan može da propadne zbog lošeg grafičkog okruženja. Kada vaš šef ili, ne daj bože, naručilac zapita kako napreduje projekat, malo će vredeti vaše nabrajanje klasa i navo?enje količine napisanog koda. Pokažite im makar i nestabilno, ali profesionalno ura?eno grafičko okruženje, i odmah će uslediti aplauz.

Programeri dobro znaju da dizajn grafičkog okruženja neće sam od sebe rešiti zadatak, ali isto tako znaju da on utiče na prodaju programa. Na svu sreću, Delphi je odlična alatka za pravljenje grafičkog okruženja, kao i solidnih,

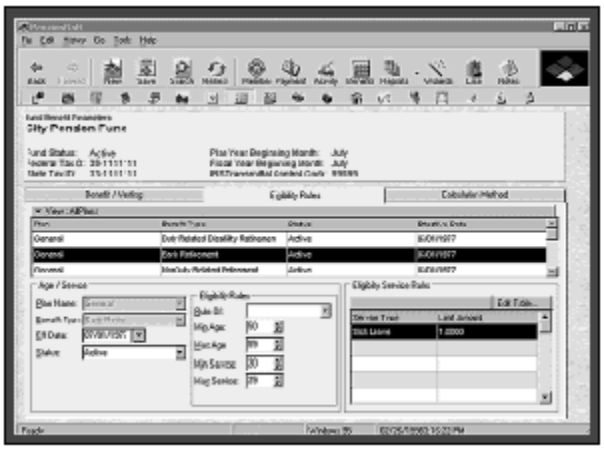
objektno orijentisanih temelja. Biblioteka vizuelnih komponenata (VCL) uglavnom sadrži izvorni kôd Object Pascala. Isti kôd kojim je napravljen Delphi na raspolaganju vam je za pravljenje aplikacija u njemu.

Delphi Super Page na Internet adresi http://delphi.icm.edu.pl/ u vreme pisanja ovog teksta sadržala je 5796 datoteka sa komponentama. To je pravo mesto za nalaženje potrebnog koda.

SAVET

Grafičko radno okruženje sadrži vizuelne efekte koji se postižu kombinovanjem koda i podataka iz različitih resursa, kao što su slike, video i audio sekvence, fontovi i boje. Pošto je Delphi objektno orijentisan jezik, već postoji možda desetine hiljada komponenata pisanih u izvornom kodu, a još mnogo će ih biti napravljeno. Delphijevo okruženje se dovoljno lako koristi, pa bi i umetnik (koji nije programer) mogao u njemu da za vašu aplikaciju pravi remek-dela. Jedan primer privlačnog grafičkog okruženja vidite na slici 2.1. To je Web strana za softverski proizvod PensionGold, napravljena u Delphiju. Kada se uzme u obzir važnost vizuelnih efekata, prosto je neverovatno koliko malo kompanija zapošljava profesionalne umetnike za dizajniranje grafičkog okruženja.

Slika 2.1 Web strana PensionGold firme LRS napravljena u Delphiju (http://www.lrs.com).



Ako sprovedete postupak koji sledi, upoznaćete novu komponentu u Delphiju, *TWebBrowser* .

U koraku 9, kada pokrenete demonstracioni program, moraćete da budete povezani na Internet da biste mogli da pristupite Web strani.

NAPOMENA

1. Pokrenite Delphi 6. Automatski će se otvoriti nov projekat.
2. Dovedite podrazumevani obrazac u prednji plan. (Pomoću tastera F12 prebacujete se na obrazac ili na editor koda.)
3. Na paleti komponenata pritisnite jezičak kartice Internet.

4. Pritisnite dvaput komponentu *TWebBrowser* ; time ćete je postaviti na obrazac.
5. Pritisnite taster F11 da biste otvorili Object Inspector.
6. U biraču objekata (Object Selector) proverite da li je izabran WebBrowser1. Vrednost svojstva Align Property promenite u Align to the Client (alClient).
7. U Object Inspectoru izaberite komponentu TForm.
8. Dok je izabran objekat Form1, pritisnite jezičak kartice Events.
9. Pritisnite dvaput doga?aj OnCreate i upišite sledeći kôd:

WebBroker1.GoHome;

1. Pritisnite taster F9.

Kada sa svog računara pokrenem ovaj demonstracioni program, na?em se na adresi http://www.softconcepts.com, na matičnoj strani moje firme prikazanoj na slici 2.2. Dobijate čitač Weba samo pomoću jedne komponente i jednog programskog reda. Iako na tržištu možda i nema mesta za nove čitače, neka ovaj samo ilustruje šta sve mogu komponente.

Slika 2.2 Nova komponenta TWebBrowser učitava Web stranu sa adrese http://www.softconcepts.com

Odlična knjiga Alana Coopera koju je izdao Sams, Pacijenti su preuzeli ludnicu (The Inmates are Running the Asylum) govori o uticaju ljudskog faktora na projektovanje. U njoj nećete naći ništa o pravljenju komponenata i kodiranju, ali ćete naći strategije projektovanja softvera koji ljudi žele da koriste.

NAPOMENA

Razmislite o tome da angažujete umetnika koji bi "ispolirao" grafičko okruženje vaše aplikacije. Najmanje što možete jeste da upotrebite potklase komponenata da biste dobili dosledan izgled uobičajenih vizuelnih elemenata, kao što su dugmad, panoi, boje, fontovi i bit mape.

Procedure i funkcije

Nijedan Windowsov program ne može bez programskog koda. Najčešće ćete ga naći u procedurama i funkcijama. U svetu objektno orijentisanog programiranja

za funkcije i procedure koje predstavljaju članove klasa koristi se opšti izraz metoda.

Argumenti se prosle?uju procedurama prema ustanovljenim pravilima pozivanja procedura. Na primer, argumenti se mogu prosle?ivati preko registara procesora. U Delphiju se to naziva pravilom pozivanja registra. Kada ceo program pišete u Delphiju, ne morate da brinete o pravilima pozivanja. Od vaših potreba zavisi kada ćete upotrebiti neku od direktiva iz tabele 2.1. Pravila pozivanja odre?uju redosled kojim se parametri prosle?uju procedurama.

Bezbrojne primere korišćenja direktiva iz tabele 2.1 naći ćete u samom Delphiju ako odre?enu direktivu potražite koristeći opciju Find in Files. Direktiva se uvek stavlja na kraj.

TABELA 2.1 **Direktive koje menjaju redosled pozivanja argumenata, tj. pravila pozivanja**

Direktiva	**Redosled parametara**	**Parametri se prosle?uju registru procesora**
register	sleva udesno	Da
Pascal	sleva udesno	Ne
cdecl	zdesna ulevo	Ne
stdcall	zdesna ulevo	Ne
safecall	zdesna ulevo	Ne

Parametri se prosle?uju u adresni prostor steka ili u registre procesora. Redosled kojim se pozivaju i mesto gde će se podaci pri prosle?ivanju naći zavise od programskog jezika i upotrebljenog pravila pozivanja. Parametri se prosle?uju proceduri po redosledu pojavljivanja ili po obrnutom redosledu. Ako je biblioteka DLL pisana na jeziku C ili C++, parametri se prosle?uju zdesna ulevo. Kada u Delphiju deklarišete proceduru, treba da dodate i direktivu *cdecl* da biste Delphiju saopštili da izvrne redosled parametara koje očekuje. Windowsove API procedure koriste redosled *stdcall* i *safecall* ; prema tome, kada pozivate Windowsove API procedure, treba da upotrebite ove direktive. Primere potražite u nastavku poglavlja, u odeljku "Pozivanje Windowsovih API procedura".

Windows je operativni sistem zasnovan na porukama

Programeri koji žele da iskoriste sve mogućnosti Windowsa moraju da razumeju arhitekturu operativnog sistema zasnovanu na porukama. Takvi operativni sistemi liče na PTT usluge. Pošiljalac napiše pismo i baci ga u poštansko sanduče. Poštanski službenik pokupi sva pisma iz sandučića ne brinući o njihovoj sadržini, i odnese ih u službu koja će ručno ili mašinski, razvrstati poštu prema poštanskim brojevima. Pisma se dalje dostavljaju primaocima prema adresi. Poenta je u tome da, dokle god pošiljka ne liči na nešto opasno, niko od onih koji s njom dolaze u dodir ne brine o sadržaju pošiljke, osim primaoca.

Pošiljalac i primalac poruke brinu o njenom sadržaju, ali prenosioci poruke brinu samo o tačnom dostavljanju.

Windows radi na sličan način - samo malo drugačije. To više liči na vojnike koji su se okupili oko "poštara" željno očekujući da čuju i svoje ime. Zasada nema načina da Windows brine o sadržaju poruke pošto softver koji bi to radio uglavnom još nije napisan. Ako želite da pišete programe za Windows, dobro bi bilo da znate kako da odgovorite na poruke. Dobro je što Delphi već zna kako da se sna?e sa većinom poruka, a još bolje je to što ćemo u narednom odeljku pokazati kako da pišete metode za rad sa porukama i obradu doga?aja.

Obra?ivači doga?aja povezuju Windows sa programima pisanim za njega

Doga?aj je, kako bi rekla deca, kada se nešto dogodi. Pritiskanje tastera miša je doga?aj. Uradite to i nešto će se desiti. Pa dobro, najpre će električni impuls uzrokovati izvršavanje funkcija prekida 0 · 09 i 0 · 16 u BIOS-u, a zatim će o tome biti obavešten Windows. Ono što korisnici čine kada rade na računaru i ono što sam računar čini dok radi, naziva se doga?ajima. Očigledno je da Windows mora da ima mehanizam kojim će odgovoriti na doga?aj. Doga?aj se desi, Windows utvrdi šta je to, oformi poruku i stavi je na raspolaganje svim programima sposobnim da je prime. Eto kako deluje operativni sistem koji je zasnovan na doga?ajima i porukama.

Procedure povratnog pozivanja

Svi podaci u računaru predstavljeni su brojevima koji odgovaraju odre?enim stanjima poluprovodničkih elemenata. Jasno je da su celi brojevi - brojevi, ali su i tekstualni nizovi brojevi, a i lokacije procedura u memoriji računara izražavaju se brojevima. Svaki podatak ima svoju adresu u računaru. Iz toga sledi da svemu možete da pristupite ako znate gde se nalazi, tj. ako znate ispravnu adresu.

Procedura povratnog pozivanja (engl. callback) jeste procedura koja se poziva navo?enjem adrese. Pozivalac u trenutku prevo?enja ne mora obavezno da zna koju proceduru treba da zove. On mora da zna samo njen potpis. Poziv se upućuje toj proceduri, a pozivalac pamti adresu koja je predstavlja. Takav postupak omogućava da u trenutku izvršavanja aplikacije bude na raspolaganju odre?ena procedura povratnog pozivanja koja se tokom rada aplikacije može i promeniti. Proces koji poziva proceduru povratnog pozivanja već je preveden i zna kako da se pozove sa odre?enim interfejsom, ali ne zna unapred njeno simboličko ime. Pomislite i sami: kako Windowsov API može da zna ime funkcije koju upravo pišete. Naravno da ne može, ali može da zacrta kostur procedure. Kada dobije adresu koja ukazuje na stvarnu proceduru koja ima taj kostur, pozivalac može da je pozove iako o njoj unapred ne zna ništa. Procedure povratnog pozivanja neophodne su za programiranje u Windowsu. (Više obaveštenja o ovome naći ćete u poglavlju 6, gde govorimo o proceduralnim tipovima.)

Obra?ivači doga?aja

Obra?ivačem doga?aja, ili procedurom za obradu doga?aja (engl. event handler) nazivamo proceduru koja reaguje na doga?aje. Ta procedura je povezana sa porukom na sledeći način. Prvo Windows ili neki drugi program saznaju za

doga?aj. Oni generišu poruku. Obra?ivač poruka (engl. message handler) prima poruku; on ima adresu proceduralnog tipa, identifikator doga?aja (engl. event handle), i poziva obra?ivač doga?aja procedurom povratnog pozivanja. Ovaj postupak je prikazan na slici 2.3.

Dijagram je prikazan u linijskom obliku. Od čovečuljka na levom kraju - osobe koja pritiska tastaturu, sledite strelice udesno i naniže. Vertikalne isprekidane linije ukazuju na objekat za koji je vezana odre?ena aktivnost. Na samom levom kraju dijagram prikazuje osobu kako pritiska taster. Na dijagramu je tastatura bukvalno objekat, a KeyPress metoda tog objekta. BIOS predstavlja objekat, a dijagram prikazuje kako tastatura poziva njegovu metodu InterpretKeyState. Dijagram približno prikazuje sve učesnike u procesu odgovaranja na pritisak tastera. Vertikalna linija pripojena strelici predstavlja element koji se ponaša na imenovan (poznat) način - tekst iznad strelice.

NAPOMENA

Slika 2.3 Pojednostavljen grafički prikaz arhitekture Windowsovog rada sa porukama izazvanim događajima koji počinju aktivnošću na tastaturi

Delphi samo traži da u poslednjem koraku dodate kôd: vi treba da napišete proceduru za doga?aj, a prazna procedura se može generisati ako dvaput pritisnete karticu Events u Object Inspectoru. Sva unutrašnja složenost svojstvena Windowsu skrivena je Delphijevom arhitekturom. Na sreću, niste ograničeni samo na pisanje koda za obradu doga?aja. U naprednom programiranju ponekad je korisno ako napišete sopstveni obra?ivač poruka koje stižu od Windowsa ili definišete sopstvene procedure i obra?ivače doga?aja. (U poglavlju 6 potražite više podataka o labavom sprezanju i obra?ivačima poruka, odnosno doga?aja.)

Objekti samo olakšavaju programiranje - ne čine ga lakim

Delphi olakšava programiranje aplikacija za Windows, ali programiranje i dalje nije lako. Object Pascal prikriva neke složene elemente programiranja za Windows, mada su oni i dalje dostupni. Odgovaranje na spoljne doga?aje zahteva da u obra?ivač doga?aja unesete samo nekoliko programskih redova, ali ako želite da proširite listu poruka koje klasa može da primi, možete i to.

Pozivanje Windowsovih API procedura

Windowsovom API-ju možete da pristupite mnogo lakše iz Delphija nego iz drugih programskih jezika. Delphi ima i programske jedinice sa funkcijama koje omotavaju uobičajene API procedure, što omogućuje da koristite procedure Pascala a da ne morate da uvodite API deklaracije. Me?utim, ponekad će vam zatrebati procedure iz dinamički povezanih biblioteka koje nisu deo Windowsa.

Zbog toga je u principu dobro da znate kako se deklarišu i koriste DLL procedure.

Ako vam zatreba neka API metoda, setite se da su takve metode već deklarisane u programskim jedinicama, kao što je Windows.pas. Nema potrebe da ih ponovo deklarišete.

Izvršne datoteke ili dinamički povezane biblioteke?

Aplikacije se pojavljuju u dva osnovna oblika: kao izvršne datoteke i kao dinamički povezane biblioteke. Imena izvršnih datoteka završavaju se na .exe, a one se izvršavaju kao samostalni programi. Imena dinamički povezanih biblioteka završavaju se na .dll, a one se koriste tako što ih učitavaju druge aplikacije. Osnovna aplikacija u Windowsu izvršnog je tipa, a biblioteka procedura ili klasa predstavlja dinamički povezanu biblioteku. Model običnih objekata (COM, Common Object Model) tako?e koristi formate EXE i DLL za razmenu datoteka sa serverom aplikacija. U ovom poglavlju nećemo razmatrati COM servere nego osnovne izvršne programe i biblioteke.

Kako se poziva Windowsova API procedura

Ako pogledate izvorni poddirektorijum (u verzijama Professional i Enterprise), u koji ste instalirali Delphi 6, primetićete i direktorijum RTL. U njemu postoji više poddirektorijuma sa programskim jedinicama pisanim u Pascalu koje već sadrže deklarisane Windowsove API procedure. Vaše je samo da ime odgovarajuće programske jedinice dodate u odredbu *Uses* i da zatim funkciju pozovete na uobičajen način.

Windowsova API procedura *SendMessage* deklarisana je u odeljku sa interfejsom datoteke Windows.pas na sledeći način:

```
function SendMessage(hWnd: HWND; Msg: UINT; wParam: WPARAM;
lParam: LPARAM):
LRESULT; stdcall;
```

Navedena deklaracija pokazuje da funkcija SendMessage vraća podatak tipa *LRESULT* , a prihvata četiri argumenta, tipova *HWND* , *UINT* , *WPARAM* i *LPARAM* .

Uvezene API procedure moraju se deklarisati tako da odgovaraju implementacijama na serveru na kome su definisane. Zbog toga su gornji tipovi ponovo deklarisani i pojednostavljeni. Na primer, UINT je neoznačen ceo broj. U stvari, sve API procedure imaju notacione prefikse pošto je Microsoft glavni izvor ovih nakaznih dodataka. H znači ukazatelj (Handle), U je prefiks za neoznačen (Unsigned) broj, W označava programsku reč (Word), a L (Long) dugačak ceo broj. Ne preporučujem da ovo učite napamet; sve što vam treba pronaći ćete u sistemu pomoći.

Implementacija u stvari sadrži deklaraciju koja implementira navedenu proceduru

```
function SendMessage; external user32 name 'SendMessageA';
```

Odredba external ukazuje na ime biblioteke koja sadrži proceduru. Odredba *external user32* znači da je u pitanju biblioteka user32.dll. Ako ime *user32* potražite u datoteci Windows.pas, utvrdićete da je *user32* konstanta definisana kao ' *user32.dll'* . Pošto je Delphi deklarisao ovu proceduru, da biste je upotrebili, treba samo da Windowsovu programsku jedinicu dodate odredbi uses i da proceduru pozovete. Isprobajte sledeći primer:

1. Napravite nov projekat (koji će stvoriti nov prazan obrazac).
2. Otvorite standardnu karticu palete komponenata i pritisnite dvaput kontrolne objekte TEdit i TButton. Po jedna instanca svakog od ovih objekata pojaviće se na obrascu Form1.
3. U Object Inspectoru izaberite svojstvo Caption objekta Button1. Upišite Toggle Selection kao natpis dugmeta Button1.
4. Na obrascu dvaput pritisnite objekat Button1 da biste generisali obra?ivač za doga?aj OnClick dugmeta Button1. Proceduri za obradu doga?aja dodajte ovaj kôd:

```
1. procedure TForm1.Button1Click(Sender: TObject);
2. const
3. StartPosition : Integer = 0;
4. EndPosition : Integer = -1;
5. begin
6. Edit1.SetFocus;
7. StartPosition := Not StartPosition;
8. EndPosition := Not EndPosition;
9. SendMessage(Edit1.Handle, EM_SETSEL, StartPosition,
EndPosition);
10. end;
```

Prvi i poslednji red dodaće sam Delphi. Redovi 1 i 2 definišu tipizirane konstante čija je uloga slična ulozi statičnih promenljivih u jeziku C++. One zadržavaju istu vrednost izme?u uzastopnih poziva proceduri. *Edit1.SetFocus* smešta u fokus objekat Edit1 pošto dugme Button1 dolazi u fokus kada ga pritisnete. Redovi 6, 7 i 8 promenljivama *StartPosition* (početno stanje) i *EndPosition* (krajnje stanje) menjaju vrednosti (0 u -1 i obrnuto). U 9. redu poziva se Windowsova API funkcija *SendMessage* koja je deklarisana u datoteci Windows.pas. Već smo pomenuli da svaki potomak tipa *TWinControl* sadrži Windowsov ukazatelj (Handle) koji odgovara prvom argumentu. *EM_SETSEL* je unapred definisana Windowsova poruka. Ako promenljiva *StartPosition* ima vrednost -1, a promenljiva *EndPosition* vrednost 0, tekst izlazi iz fokusa. Zamenite vrednosti ove dve promenljive i sav tekst će biti izabran.

Samo u datoteci Windows.pas ima 30.000 programskih redova sa deklaracijama. Windowsov interfejs za programiranje aplikacija (API) već je ogroman i stalno postaje veći. Moraćete da koristite priručnu literaturu da biste

saznali čega sve tu ima. Provedite malo vremena u razgledanju API-ja pre nego što počnete da implementirate nove procedure. Ako neke procedure nema u Delphiju, verovatno se nalazi na nekom drugom mestu.

Deklarisanje API procedure

Svaka biblioteka se po pravilu naziva API. Ako ne možete da prona?ete izvornu programsku jedinicu, npr. Windows.pas, koja sadrži deklaraciju za odre?enu API proceduru, onda proceduru deklarišite sami. Sledi primer deklaracije procedure *SendMessage* za slučaj da datoteka Windows.pas nije dostupna.

function SendMessage(hwnd : Longword; MSG : *Longword;*
wParam : Longint; lParam : longint) : longint;
stdcall; external 'user32.dll' name 'SendMessageA';

Korišćenje odredbe *external* za učitavanje metode iz biblioteke istovremeno sa učitavanjem programske jedinice naziva se rano razrešavanje (engl. early binding). Suprotno od toga je kasno razrešavanje (engl. late binding) gde je neophodan izričit poziv funkciji LoadLibrary.

Osnovna sintaksa zahteva opisivanje tipa procedure, imena i argumenata. U programskoj jedinici Windows.pas u stvari se koriste Windowsovi tipovi podataka, a funkcija deklariše ovako:

function SendMessage(hWnd: HWND; Msg: UINT; wParam: WPARAM;
lParam: LPARAM): LRESULT; stdcall;

U navedenoj deklaraciji umesto Windowsovih tipova korišćeni su odgovarajući tipovi Pascala. Na primer, Windowsov tip *LParam* ekvivalentan je tipu *Longint* u Object Pascalu. Na samom kraju deklaracije za uvoženje nalaze se direktive. Direktiva *stdcall* za proceduru *SendMessage* nalaže pravilo pozivanja zdesna ulevo, umesto sleva udesno, kako je u Pascalu, a argumenti se ne prosle?uju registrima procesora. Direktiva external kaže da se procedura *SendMessage* nalazi u Windowsovoj datoteci user32.dll, a direktiva name ukazuje na stvarno ime funkcije (deklarisano u datoteci user32.dll) - *SendMessageA* .

Ako API funkciju želite da uključite u interfejs svoje programske jedinice i time drugim jedinicama omogućite da je pozivaju a da ne moraju da je ponovo deklarišu, onda deklaraciju možete da podelite u deo sa interfejsom i deo sa implementacijom. Ako funkciju želite da dodate u deo vaše programske jedinice koji sadrži interfejs, onda u njemu neka ostane sve osim odredbi external i name.

function SendMessage(hwnd : Longword; MSG : Longword;
wParam : Longint; lParam : longint) : longint; stdcall;

U deo sa implementacijom unesite kôd kojim se implicitno poziva API.

function SendMessage; external 'user32.dll' name 'SendMessageA';

Vaša procedura *SendMessage* spolja liči na bilo koju drugu proceduru. Upamtite da nema potrebe da deklarišete većinu Windowsovih API procedura;

firma Inprise je to već uradila umesto vas. Me?utim, bez obzira na to ko to radi, tehnika je uvek ista.

Učitavanje biblioteka u hodu

U prethodnom odeljku ilustrovali smo implicitno učitavanje datoteka. Deklaracija koja sadrži tekst external 'user32.dll' implicitno nalaže prevodiocu da učita datoteku user32.dll u trenutku izvršavanja aplikacije. Pretpostavimo da vam neka mogućnost treba samo s vremena na vreme. Možda ne biste želeli da opteretite sistem sa više DLL biblioteka koje su sve vreme u memoriji. Ako želite zaista da upravljate učitavanjem DLL datoteka, onda morate da upotrebite procedure *LoadLibrary* , *FreeLibrary* i *GetProcAddress* .

Procedura *LoadLibrary* na vaš zahtev će učitati odre?enu datoteku. Procedura *FreeLibrary* će je ukloniti iz memorije, a procedura *GetProcAddress* će dinamički pronaći ukazatelj na odre?enu proceduru. Ako biblioteku želite da učitate procedurom *LoadLibrary* , onda ne treba da je deklarišete naredbom external. Umesto toga, deklarišite tip čiji je otisak isti kao i otisak procedure koju želite da učitate. Kada poželite da koristite proceduru, pozovite *LoadLibrary* , onda *GetProcAddress* da biste pronašli adresu procedure u biblioteci i na kraju, kada ste s njom uradili šta ste želeli, uklonite je iz memorije procedurom *FreeLibrary* .

Izričito učitavanje biblioteka i procedura naziva se kasno razrešavanje. Adresa procedure je nepoznata u trenutku prevo?enja programa. (U prethodnom odeljku je primer ranog razrešavanja.)

Sledeći kôd ilustruje elemente potrebne za učitavanje dinamički povezane biblioteke u trenutku izvršavanja i za dobijanje adrese procedure od servera.

```
unit USendMessage;
// USendMessage.pas Ilustruje dinamičko učitavanje API-ja
// i inicijalizovanje procedure
// Copyright (c) 2000. All Rights Reserved.
// by Software Conceptions, Inc. Okemos, MI USA (800) 471-5890
// Written by Paul Kimmel
interface
uses
Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,
StdCtrls, Menus;
type
TSendMessage = function (hwnd : Longword; MSG : Longword;
wParam : Longint; lParam : longint ) : longint; stdcall;
TForm1 = class(TForm)
Edit1: TEdit;
Button1: TButton;
procedure Button1Click(Sender: TObject);
```

```
procedure FormCreate(Sender: TObject);
procedure FormDestroy(Sender: TObject);
private
{ Private declarations }
Instance : Longword;
MySendMessage : TSendMessage;
public
{ Public declarations }
end;
var
Form1: TForm1;
implementation
{$R *.DFM}
procedure TForm1.Button1Click(Sender: TObject);
const
StartPos : Integer = 0;
EndPos : Integer = -1;
begin
Edit1.SetFocus;
StartPos := Not StartPos;
EndPos := Not EndPos;
MySendMessage(Edit1.Handle, EM_SETSEL, StartPos, EndPos);
end;
procedure TForm1.FormCreate(Sender: TObject);
begin
Instance := LoadLibrary('user32.dll');
if( Instance <> 0 ) then
MySendMessage := GetProcAddress( Instance, 'SendMessageA');
end;
procedure TForm1.FormDestroy(Sender: TObject);
begin
FreeLibrary(Instance);
end;
end.
```

U odeljku *Type* definiše se proceduralni tip *TSendMessage* identičan otisku procedure *SendMessage* . (Detalje potražite u poglavlju 6 u odeljku "Napravite sopstvene proceduralne tipove", a ovde samo pokušajte da shvatite smisao.) U privatnom odeljku klase *TForm* deklariše se promenljiva Instance tipa *Longword* . Deklariše se i promenljiva *MySendMessage* tipa *TSendMessage* . Ukazatelj Instance biće ukazatelj na datoteku user32.dll koji je vratila procedura *LoadLibrary* . Otvorite Object Inspector i izaberite obrazac Form1. Na kartici Events Object Inspectora pomoću dvostrukog pritiska mišem generišite procedure za obradu doga?aja *FormCreate* i *FormDestroy* . U navedenom kodu datoteka user32.dll učitava se kroz metodu *FormCreate* , a briše kroz metodu *FormDestroy* . Procedura za obradu doga?aja *FormCreate* - procedura konstruktora - učitava biblioteku user32.dll i nalazi adresu procedure

SendMessage , a zatim je dodeljuje promenljivoj *MySendMessage* . Za sve ono što vama treba, *MySendMessage* predstavlja funkciju *SendMessage* iz biblioteke user32.dll.

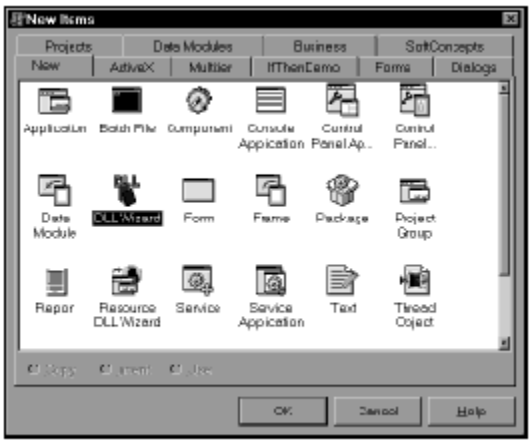
Postoji više API-ja koji mogu da reše gotovo sve vaše probleme, ali imajte na umu da je učitavanje DLL biblioteka, bilo implicitno, bilo eksplicitno, prevazi?ena tehnologija. COM je uspostavio protokol za korišćenje servera aplikacija koji se koristi isto tako lako, ako ne i lakše. O ovoj tehnologiji možete da na?ete više obaveštenja u poglavlju 12 i dodatku C.

NAPOMENA

Pravljenje dinamički povezane biblioteke

Dinamički povezana biblioteka predstavlja jedan od tipskih projekata u Delphijevom skladištu. Ako želite nov projekat biblioteke, u meniju redom pritisnite stavke File, New, Other i na kartici New okvira za dijalog New Items, prikazanog na slici 2.4, izaberite stavku DLL Wizard. Najupadljivija razlika izme?u biblioteke i izvršne datoteke jeste to što izvorna datoteka projekta - pritisnite tastere Alt+P, V da biste videli izvorni kôd projekta - sadrži direktivu library, umesto direktive program. Direktiva library nalaže prevodiocu da aplikaciju prevede u DLL datoteku. Sledeći kôd prikazuje kako možete da napravite sasvim jednostavnu biblioteku.

Slika 2.4 Komanda File, New, Other otvara okvir sa dijalogom novih stavki. Dvaput pritisnite stavku DLL Wizard i započnite DLL projekat.




```
library TestDll;
uses Sharemem, SysUtils, Dialogs, Classes;
{$R *.RES}
Procedure Test;
begin
ShowMessage('Ovo je proba!');
end;
exports
Test;
begin
```

end.

Dodati su samo procedura *Procedure Test* koja prikazuje poruku i deklaracija *exports* . Ako vam treba išta složenije od ovoga, onda morate da umetnete programsku jedinicu i u nju smestite kôd. Naredba *exports* naznačava procedure koje će biti deo interfejsa DLL datoteke, tj. procedure koje će biti dostupne drugim aplikacijama. U biblioteci možete da imate koliko god hoćete procedura, ali u njen interfejs smestite samo one koje želite da izvozite, tj. one koje će moći da pozivaju drugi programeri. Kada želite da generišete probnu datoteku TestDll.dll, pritisnite tastere Alt+P, B. Posle toga treba da generišete probnu aplikaciju.

Testiranje dinamički povezane biblioteke

Ako želite da testirate biblioteku, morate da napravite nov projekat. Æelimo aplikaciju podrazumevanog izvršnog tipa. Dodajte deklaraciju koja implicitno učitava biblioteku i uvozi proceduru. Oslanjajući se na prethodni primer, to ćete postići jednim programskim redom:

```
Procedure Test; external 'TestDll.dll';
```

U ovoj fazi procedura iz biblioteke može se pozvati kao bilo koja druga procedura. Poziv glasi:

```
Test;
```

Biblioteku koju napravite, kao i probni projekat, možete pomoću Project Managera da dodate istoj projektnoj grupi. Pritisnite tastere Alt+V, P da biste iz menija View otvorili Project Manager. Pritisnite Project Manager desnim tasterom miša i iz priručnog menija (slika 2.5) izaberite dodavanje novog (Add New Project) ili nekog već postojećeg (Add Existing Project...) projekta.

Slika 2.5 Pomoću Project Managera istoj projektnoj grupi možete da dodate više projekata. To će vam omogućiti da u grupnim projektima lakše otklanjate greške.

Kada su biblioteka i kôd za proveravanje u istoj projektnoj grupi, onda možete da se krećete kroz obe aplikacije prelazeći granicu DLL procesa, baš kao da se radi o jedinstvenoj aplikaciji. U daljem tekstu ćemo objasniti još nekoliko momenata na koje treba da obratite pažnju kada pravite dinamički povezane biblioteke; sa praktičnog stanovišta to samo znači da treba da proširimo kôd.

Prosle?ivanje Pascalovih znakovnih nizova dinamički povezanim bibliotekama

Pascal koristi sopstveni format znakovnih nizova. Windowsov API koristi format ASCIIZ (niz koji se završava nulom) čija se dužina odre?uje brojanjem znakova sve dok se ne do?e do nule. Pascal čuva podatke o dužini znakovnog niza u skrivenom prefiksnom polju, nizovi se prebrojavaju po referenci, a memoriju za nizove Pascal dodeljuje dinamički; prema tome, ako želite da prosledite Pascalove znakovne nizove preko DLL granice, morate kao prvu programsku jedinicu u biblioteci da navedete datoteku Sharemem.pas i projektnu odredbu *Uses* , kao što je prikazano u listingu sa početka ovog odeljka. Sharemem.pas je programska jedinica sa interfejsom ka datoteci Borlndmm.dll (programu za upravljanje memorijom) koju morate da isporučujete zajedno sa svojom aplikacijom ukoliko koristite datoteku Sharemem.pas.

Nije neophodno da koristite datoteku Sharemem.pas i da zbog toga morate sa aplikacijom da isporučujete i biblioteku Borlndmm.dll. Iako se Pascalovi znakovni nizovi lakše koriste, možete da se opredelite za nešto lošije rešenje - tip PChar. Tada vam neće trebati datoteke Borlndmm.dll i Sharemem.pas, ali ćete morati više da se pozabavite kodom.

Potrebu za datotekama Sharemem.pas i Borlndmm.dll možete da umanjite ako u procedurama kojima se prosle?uju nizovi za argumente i za vrednost koju procedura vraća upotrebite tip PChar. Promenljivu tipa PChar možete da upotrebite u situacijama u kojima se očekuje znakovni niz, ali je lakše ako se držite Pascalovih znakovnih nizova.

Pravljenje datoteke deklaracija

Kada pravite biblioteku, imajte na umu da će je koristiti druge aplikacije. Na kraju krajeva, biblioteka tome i služi. Programska jedinica za uvoženje jeste jedinica sa izvornim kodom koja se koristi za deklarisanje procedura iz biblioteke u interfejsu i njihovo "implementiranje" tako što se kao spoljne deklarišu u delu sa implementacijom. Programska jedinica za uvoženje omogućava da proceduru iz spoljne biblioteke pozovete kao i bilo koju drugu proceduru. Ako napravite programsku jedinicu za uvoženje i isporučite je programerima zajedno sa svojom bibliotekom, time ćete im omogućiti da vašu biblioteku lako koriste u programiranju. Programska jedinica sa deklaracijama za biblioteku TestDLL trebalo bi da sadrži sledeće redove:

```
unit UDeclares;
interface
Procedure Test;
implementation
Procedure Test; external 'TestDLL.dll';
end.
```

Programsku jedinicu sa deklaracijama pravite kada komandom New iz menija File napravite novu jedinicu, zatim je ubacite u odeljak sa interfejsom, a u odeljak sa implementacijom unesete definicije, baš kao u prethodnom primeru. Ako želite, možete da dodate deklaracije i drugih procedura.

Definisanje klasa i instanciranje objekata

Klase su kamen temeljac objektno orijentisanog programiranja. One su sačinjene od podataka i procedura. Kada podaci spadaju u neku klasu, pravilo nalaže da se klasa smatra poljem. Procedure definisane u klasama nazivaju se metode. Sve što je definisano unutar klase može se zvati atribut. Novi izrazi u terminologiji klasa - svojstvo, polje (umesto podaci) i metoda (umesto procedura) - uvedeni su da označe da podaci i procedure klasa nisu isto što i podaci i procedure izvan klasa. Da biste mogli da ih pravilno koristite, potrebna su dodatna objašnjenja.

U Delphiju možete da pišete strukturiran kôd, ali to nije dovoljno da Delphi u potpunosti iskoristite. Object Pascal je korišćen za implementaciju Delphija. Arhitektura Delphija je objektno orijentisana, a osnovna klasa mu je *TObject* . Svaka klasa u Delphiju potiče od klase *TObject* . (Dodatna objašnjenja potražite u poglavlju 3.) U ostatku ovog poglavlja naučićete sve što je potrebno za pisanje dobrog i izražajnog softvera.

Osnovna sintaksa klasa

Deklaracija svake klase sledi osnovnu sintaksu. Gramatičko pravilo možete da prona?ete u sistemu pomoći, ali za deklarisanje bilo koje klase potrebna su barem dva programska reda, kao u sledećem primeru:

```
TMyClass = class
end;
```

U gornjem listingu definiše se klasa *TMyClass* koja nema atribute. Sledeća definicija radi isto:

```
TMyClass = class(TObject)
end;
```

Prvi primer ilustruje osnovnu sintaksu za deklarisanje klase. Drugi, pak, ilustruje osnovnu sintaksu za deklarisanje potklase, odnosno nasle?ivanje. U gornjim primerima dobijaju se identične klase jer su u Delphiju sve klase potklase klase *TObject* .

Klasa *TObject* ima metodu klase *ClassName* koja vraća ime klase u obliku znakovnog niza. Ta metoda se koristi za razrešavanje tipova tokom izvršenja (engl. runtime type identification, RTTI), što je olakšano ukoliko su sve klase potomci klase *TObject* . Kada referencirate *TMyClass* , npr. u kodu,

```
ShowMessage(TMyClass.ClassName);
```

otvoriće se okvir za dijalog sa tekstom *TMyClass* , imenom klase. Poziv proceduri *ShowMessage* prikazaće isti tekst, bez obzira na to da li ste klasu definisali na prvi ili na drugi način, što ukazuje na istog pretka, klasu *TObject* . Dalji primer ovoga je sledeći mali program u kome se operator *is* koristi za proveravanje porekla klase.

```
var
MyClass : TMyClass;
begin
MyClass := TMyClass.Create;
if( MyClass Is TObject ) then
```

```
ShowMessage('MyClass je TObject');
MyClass.Free;
end;
```

Dve verzije klase *TMyClass* su identične, mada to ne bi bio slučaj ako bi druga verzija bila potklasa neke od drugih klasa koje postoje pored klase *TObject* . Č;ak i tada bi test (*MyClass Is TObject*) iz navedenog programa dao tačan rezultat i poruka bi se prikazala. Upamtite da sve klase vode poreklo od klase *TObject* .

Ako u Delphiju pri deklarisanju klase ne navedete pretka, onda će deklarisana klasa podrazumevano naslediti klasu *TObject* .

Deklarisanje unapred

Do sada prikazani primeri odnosili su se na sintaksu za definisanje klasa. Izraz class koristi se još u nekim slučajevima. (Jedna njegova upotreba biće objašnjena u poglavlju 7, u odeljku "Metode klasa ili statične metode".) Postoji i sintaksa za deklarisanje unapred, koja uvodi ime pre stvarne definicije klase. Ona glasi:

```
TMyClass = class;
```

Deklarisanje unapred je korisno kada dve klase, A i B, u istoj programskoj jedinici ukazuju jedna na drugu. Jedna od njih mora da bude prva, pa je deklarišite unapred da biste izbegli ovu zamku. U našem primeru, A ima atribut tipa B, a B ima atribut tipa A. Klase A i B imaju me?uzavisnu definiciju. Kada nastane ovakva situacija, deklarišite prvu klasu unapred, a definišite je kasnije u odeljku sa interfejsom, kao u sledećem primeru:

```
interface
type
TClassA = class; // deklarisanje unapred
TClassB = class // definicija klase
ClassA : TClassA; // uvo?enje A pre nego što je definisano
end;
TClassA = class // definicija klase
ClassB : TClassB;
end;
```

Deklarisanjem unapred uvodi se ime klase, što prevodiocu omogućuje da razreši referenciranje klase A u klasi B pre nego što se definiše klasa A.

Dvojnici klasa

Rezervisana reč *class* koristi se i za nasle?ivanje bez novih atributa, znači za definisanje dvojnika klase (engl. class aliasing). Na primer, definišite nov tip klase *Exception* ne dodajući nikakve nove atribute:

```
type
EMyClass = class(Exception);
```

Klase Exception po konvenciji imaju prefiks *E* . Uklonite *E* i imaćete zgodno ime za instancu izuzetka.

U prethodnom kodu definiše se klasa *EMyClass* kao naslednik klase *Exception* . Pošto klasa *EMyClass* ne proširuje, niti menja ponašanje klase *Exception* , nije potrebna naredba *end* .

Definisanje metaklasa ili referenci na klase

Rezervisana reč *class* koristi se zajedno sa rezervisanom rečju *of* da bi se definisala metaklasa. "U osnovi, metaklasa je klasa klase, koncepcija koja nam omogućava da sa klasama radimo kao sa objektima [Booch, 108]". Definicija metaklase izgleda ovako:

```
type TClass = class of TSomeType;
```

Tipovi metaklasa mogu da budu korisni kada je u trenutku prevo?enja nepoznat stvarni tip klase. (Više objašnjenja potražite u poglavlju 7.)

Definisanje stanja

Deo odgovornosti klase je da skicira šablon za svoje instance. Atributi koji definišu stanje su atributi podataka. Ako imamo klasu *Dog* (pas), atributi koji dobro opisuju konkretnog psa mogu da budu boja, rasa, pol i težina. Za iznalaženje ispravne apstrakcije zadužena je analiza, a ona zavisi od prirode problema. U našem jednostavnom primeru za definisanje klase pas biće dovoljna četiri atributa.

Odre?ivanje pravih atributa prema vrsti problema predstavlja izazov objektno orijentisanog projektovanja. Za praktično definisanje stanja Attributes potrebno je da atributu dodelimo ime i odgovarajući tip podataka. Za klasu *TDog* iz našeg primera to bi moglo da izgleda ovako:

```
TDog = class
Color : TColor;
Breed : String;
Gender : String;
Weight : Double;
end;
```

Imena su potpune reči koje opisuju vrstu podataka, a tipovi treba da su smisleni. *Color := clYellow* (boja - žuta), *Breed := 'Labrador'* (rasa - labrador), *Gender := 'Male'* (pol - muški), *Weight := 49.7* (težina - 49,7). Subjektivnom procenom možemo da utvrdimo da tipovi imaju smisla. Razmotrimo, me?utim, tip *TColor* . To je unapred definisan tip koji obuhvata i članove *clTeal* (guščijeplava boja) i *clButtonText* (boja teksta na dugmetu). Koliko ja znam, ne postoje psi takve boje. Prema tome, ako ne postoji neki poseban razlog da psima damo takve neuobičajene boje, onda bi uz gornju definiciju kôd morao dodatno često da se proverava kako se u njega ne bi "uvukle" takve boje pasa.

Iz ove situacije može da nas izvuče malo inteligentnija definicija stanja. Umesto da za stanje boje upotrebimo tip *TColor* , možemo da definišemo nov nabrojani skup boja koji će obuhvatiti samo one boje koje psi mogu imati. Takav nabrojani tip mogao bi biti:

```
type TDogColor = (dcBrown, dcWhite, dcBlack, dcYellow);
```

a atribut *Color* bio bi redefinisan kao *TDogColor* . Kada upotrebite tip koji smisleno obuhvata podatke, kôd potreban za održavanje odgovarajućeg stanja biće znatno kraći. Nabrojani tipovi mogu se upotrebiti i za rasu i za pol, a za težinu upotrebite interval vrednosti da biste izbegli pse teže od 5000 kilograma ili one čija je težina manja od nule. Sve izloženo može se sažeti u sledećem kodu:

```
type
TDogColor = (dcBrown, dcWhite, dcBlack, dcYellow);
TDogGender = ( dgMale, dgBitch );
TDogBreed = ( dbLabrador, dbPoodle ); // itd
TDogWeight = 0..300;
TDog = class
Color : TDogColor;
Gender : TDogGender;
Breed : TDogBreed;
Weight : TDogWeight;
end;
```

Primenite standard "dovoljno dobro". Apstrakcije ne moraju da budu savršene, već dovoljno dobre. Kada to postignete, pre?ite na sledeći deo koda.

Druga verzija gornjeg koda mnogo je čitljivija, a smislene vrednosti podataka sadržane su u novim tipovima. Ako potrošite malo vremena na pravljenje dobre apstrakcije, to će predupredi ti pojavu grešaka i osloboditi vas pisanja mnogo koda.

Dodavanje sposobnosti

Ako podaci definišu ono što instanca klase zna, onda sposobnosti definišu šta ona može da uradi. Metode definišu mogućnosti klase. Metode su procedure i funkcije koje su deklarisane kao članovi klase. To se postiže kada se deklaracija metode smesti izme?u prvog reda naredbe *class* i rezervisane reči *end* . Evo klase *TDog* (pas) sa nekoliko odgovarajućih metoda: *Jump* (skače), *Run* (trči), *Bark* (laje), *Sleep* (spava) i *Eat* (jede).

```
TDog = class
Color : TDogColor;
Gender : TDogGender;
Breed : TDogBreed;
Weight : TDogWeight;
Procedure Jump;
```

```
Procedure Run;
Procedure Bark;
Procedure Sleep;
end;
```

Ako u izvornoj definiciji klase deklarišete metode pre polja (tj. podataka), dobićete poruku o grešci "Field definition not allowed after methods or properties". (Definisanje polja nije dozvoljeno posle definisanja metoda ili svojstava.) Da biste sprečili nastajanje ove greške, kopirajte polja na početak definicije klase, ispred svojstava.

Delphi 6 će automatski u kôd uneti deklaracije metoda pristupanja, potpune deklaracije svojstava i ispravnom sintaksom ispisati tela metoda. Da biste kôd dopunili, pritisnite klasu desnim tasterom miša. Otvoriće se priručni meni editora koda. Pritisnite u njemu stavku Complete Class at Cursor i Delphi će automatski ispisati tela metoda u odeljku sa implementacijom.

U delu sa implementacijom metode se definišu na skoro isti način kao u procedurama izvan klasa. Razlika je u tome što se ovde dodaje ime klase. Procedura *jump* definiše se na sledeći način:

```
Implementation
Procedure TDog.Jump;
begin
// ovde dolazi kôd
end;
```

U dokumentaciji Delphijevog sistema pomoći, naredbe *begin/end* nazivaju se blok.

Sve definicije procedura u delu sa implementacijom pišu se sa kodom izme?u naredaba *begin* i *end* . Osim što pripadaju klasi u kojoj su definisane, metode slede ista pravila pisanja kao i procedure, odnosno funkcije koje su izvan klasa. Svaki argument koji se može upotrebiti u proceduri, dobar je i za metodu.

Svojstvo (engl. property) je poslednji osnovni pojam koji se odnosi na definisanje klasa. Svojstva su odre?en prikaz podataka koji sledi sopstvenu sintaksu. Pogledajte odeljak "Dodavanje svojstava klasama", gde se govori o dostupnosti, važnom aspektu objektno orijentisanog projektovanja.

Pravljenje instance objekta

Pošto ste definisali klasu, nećete moći da koristite njene podatke i sposobnosti dok ne napravite njenu instancu koju zovemo objekat. (Da biste mogli da koristite svojstva i metode koje su izvan klase, morate da napravite instancu klase. O tome govorimo u poglavlju 7.) Iskaz za pravljenje instance uvek ima

isti oblik *imepromenljive := imeklase.create* . Na primer, da biste napravili instancu klase *TDog* , morate da deklarišete promenljivu tipa *TDog* i da pozovete metodu *Create* .

```
var
Dog : TDog;
begin
Dog := TDog.Create;
Dog.Color := clYellow;
// ovde dodati dalji kôd
Dog.Free;
end;
```

Poziv metodi *Create* rezerviše memoriju za instancu objekta, a metoda *Free* je osloba?a. Možda se pitate kada su definisane metode *Create* i *Free* . Setite se da sve klase imaju istog zajedničkog pretka, klasu *TObject* . Metode *Create* i *Free* definisane su u klasi *TObject* . Metoda *Create* je sposobna da rezerviše memoriju na osnovu sveukupnosti zahteva objekta, a to je memorijski prostor potreban atributima. Metoda *Free* zna kako da memoriju koju je zauzimao objekat vrati Windowsu na raspolaganje. Metoda *Create* se naziva konstruktor. Metoda *Free* će se uspešno završiti čak i kada objekat koji je poziva ne postoji. Ako takav objekat postoji, metoda *Free* će pozvati destruktora objekta, *Destroy* .

Konstruktor klase

Za svaki objekat je potreban konstruktor. Konstruktor za *TObject* definisan je u datoteci System.pas. (U poglavlju 1 rekli smo da svaka programska jedinica podrazumevano koristi datoteku System.pas.) *Constructor* je specijalna rezervisana reč koja označava da se procedura koristi za inicijalizovanje instanci klase. Pošto je *TObject* zajednički predak svih klasa, sve klase imaju podrazumevani konstruktor. (Definiciju podrazumevanog konstruktora naći ćete u deklaraciji klase *TObject* .) Konstruktori inicijalizuju stanje objekta.

Podrazumevani konstruktor nema argumenata; samo ispred poziva konstruktoru *Create* stavite ime klase, kao u gornjem primeru. Potklase mogu da redefinišu podrazumevani konstruktor, čak mogu da mu dodaju argumente. U poglavlju 4 naći ćete sve o redefinisanju podrazumevanog ponašanja konstruktora.

Destruktor klase

Podrazumevani definisani destruktor je *Destroy* . Metoda *Destroy* obavlja čišćenje koje zahteva klasa. Klasa *TObject* tako?e definiše proceduru *Free* koja proverava da li objekat postoji, a tada poziva metodu *Destroy* . Č;ak i kada objekat ne postoji, metoda *Free* ne generiše grešku.

Tabela virtuelnih metoda (engl. virtual method table, VMT) predstavlja niz procedura (koje pripadaju klasi) kojima upravlja prevodilac. Kada se metoda definiše direktivama *virtual* , *overloaded* ili *override* , ona se unosi u VMT. Kada se pozove virtuelna metoda, koristi se naročita instanca klase da bi se utvrdilo koju od raspoloživih metoda treba pozvati. VMT je tabela metoda koja olakšava polimorfno ponašanje. Bjarne Stroustrop,

autor jezika C++, daje odličan opis VMT-a u knjizi Dizajn i evolucija jezika C++ (The Design and Evolution of C++), u izdanju Addison-Wesleyja.

Destruktor *Destroy* nema argumenata; ako želite da izmenite njegov način deinicijalizacije objekata, metodu *Destroy* morate da redefinišete u potklasi.

Procedura *Free* poziva metodu *Destroy* ukoliko objekat postoji. Pogledajte definiciju iskaza *TObject.Free* da biste detaljno sagledali kôd. Slično konstruktoru, i destruktor *Destroy* može da bude redefinisan u potklasama. Tu tehniku opisujemo u poglavlju 4.

Skrivanje informacija je dobar potez

Svaki kôd je mač sa dve oštrice. Makar i da ste njegov autor, kada ga upotrebljavate, onda ste njegov korisnik. Isti odnos postoji i kada autor i korisnik nisu iste osobe. Primer prve varijante tog odnosa je pisanje koda za klase koji kasnije koristite. Primer za drugu varijantu je korišćenje biblioteke vizuelnih komponenata. Razvijanje softvera je izazovna disciplina, ali je neophodno da se složeni aspekti programiranja podele na više manjih, me?usobno nezavisnih poslova.

Pri pisanju bilo kakvog softvera postoje tri primarna cilja: dovršiti ga; neka radi ispravno; neka bude jednostavan. Začudo, veoma je teško postići bilo koji od ovih ciljeva. Dovršavanje programa zavisi od druga dva cilja. Ne možete reći da ste program dovršili ako on ne radi ispravno i ako treba još da ga doterujete. Reći da je program dovršen znači da će jedno vreme u okviru svog životnog ciklusa obavljati posao za koji je namenjen. Reći da radi ispravno znači otkriti šta korisnicima stvarno treba i obezbediti da implementacija sistema iza?e u susret tim potrebama ili da ih čak prevazi?e. Uraditi ga dobro znači biti u mogućnosti da unesete ono što ste u prvom času prevideli. Dobar program mora lako da se održava. Odličan program mora da bude proširiv. Način na koji se hvatate u koštac sa složenošću odre?uje i postizanje tri pomenuta cilja.

Specifikatori pristupa

Kada govorimo o skrivanju informacija, ne mislimo da one treba da postanu nevidljive, nego ograničeno dostupne. Ako se atributima klase ne može pristupiti, onda se oni ni na koji način ne mogu menjati, a to je kao i da ih nema. Me?utim, kada ostavimo selektivan pristup detaljima, time omogućujemo lakše rešavanje složenih situacija. Na primer, upravljanje automobilom je mnogo lakše kada su vam odmah na raspolaganju njegove komande nego kada morate prethodno da naučite svu teoriju o motorima sa unutrašnjim sagorevanjem.

Steven Pinker (W. W. Norton & Company) napisao je zanimljivu knjigu, Kako radi um (How the Mind Works), u kojoj se raspravlja o različitim načinima mišljenja i saznavanja.

Č;ovekov mozak ima izuzetnu sposobnost trajnog skladištenja informacija. Koliki je kapacitet tog trajnog pamćenja nije poznato - zna se samo da je veliki. Nasuprot tome, o privremenom pamćenju zna se poprilično, a rezultat nije baš slavan. Č;ovek ima sposobnost da privremeno upamti izme?u sedam i devet bitova podataka. Tehnikom grupisanja ova sposobnost se može povećati, mada ne mnogo. Tom tehnikom bitovi podataka se grupišu u celine i tretiraju kao novi, veći bitovi podataka. Na primer, 8 3 6 1 0 9 3 predstavlja niz od sedam pojedinačnih brojeva. Mozak sada pokušava da ih na smislen način grupiše u manje od sedam grupa. Kada od njih napravimo dve grupe, prefiks i sufiks, npr. 836-1093, sa brojevima se lakše radi. I to je sve. Postoji, naravno, granica broja podataka koji se mogu grupisati u kratkotrajnom pamćenju

Pisanje koda se odvija unutar kratkotrajnog pamćenja. Prema tome, da biste efikasno mogli da obavljate složene programske zadatke, neophodno je da ih mudro grupišete, a neke od njih i da trajno zapamtite. Dok čitate ovu knjigu, vi u svoju trajnu memoriju smeštate koncept klase u Object Pascalu. Kada budete pisali kôd za novu klasu, mislićete o onome što znate o klasama, tj. o onome što je o njima u vašem mozgu trajno uskladišteno. Za efikasno programiranje je potrebna dalja podela grupa informacija, u čemu će vam pomoći tzv. specifikatori pristupa. Object Pascal uvodi četiri takva specifikatora pomoću kojih različitim delovima klase možete da dodelite različit nivo dostupnosti. Na taj način autor klase može da razmišlja o njenom unutrašnjem ustrojstvu izdeljenom u četiri moguće grupe, a ne o kodu izvan klase. Kada budete koristili klase, treba da poznajete samo one njihove delove kojima možete da pristupite.

Javan pristup

Javan pristup važi, npr. za putnički terminal na aerodromu. U njega svako može da u?e i iz njega da iza?e bez ikakvih posledica. Bez odre?enih ograničenja pristupa, aerodromi bi bili mnogo manje bezbedni nego što jesu. Ograničenje pristupa avionu sprečava pojavu slepih putnika. Ograničenje pristupa prtljagu sprečava krijumčarenje. Kada ograničite pristup važnim delovima svojih klasa, moći ćete bolje da upravljate kodom, baš kao što aerodromski službenici ostvaruju bolju kontrolu u područjima aerodroma kojima je pristup ograničen. Svi javni atributi - upamtite da se atributi odnose na podatke i metode unutar klase - pristupačni su svima. Klasama Object Pascala pristup je podrazumevano javan, osim ako pri prevo?enju klase ili njenih predaka nije iskorišćena direktiva *$M+* . Deo klase možete izričito da označite kao javan ako unutar definicije klase smestite rezervisanu reč *public* . Svi atributi definisani izme?u rezervisane reči *public* i kraja definicije klase ili drugog specifikatora pristupa biće javni. Atributi u javnom delu klase nazivaju se javni interfejs.

Ukoliko se specifikator pristupa ne navede, identifikatorima je pristup javan, osim ako se klasa ne prevede zajedno sa izvršnom informacijom, naznačenom kroz direktivu prevodiocu *$M+* . Ako se upotrebi direktiva *$M+* , onda je podrazumevan publikovan pristup. Potklasa klase koja ima izvršnu informaciju tako?e ima izvršnu informaciju. Sve

komponente se izvode iz klase *TPersistent* koja ima izvršnu informaciju. Prema tome, pristup svim komponentama podrazumevano je publikovan. Dakle, ako u definiciji komponente ne naznačite izričito način pristupanja metodama ili podacima, svaki korisnik će direktno moći da pristupi svim unetim metodama i da menja sve unete podatke. (Pogledajte odeljak "Publikovan pristup".)

Kao što aerodromska služba mora da ograniči pristup izvesnim delovima aerodroma, tako i autor programa treba da ograniči pristup izvesnim aspektima svojih klasa. Na primer, putnici svoj prtljag mogu da podignu na mestu koje je za to odre?eno, ali nijedan putnik u načelu ne može da pristupi skladištu prtljaga na aerodromu ili u avionu. Takav sistem radi dobro. Ista pravila važe i za klase. U javni interfejs stavite samo ono što želite da bude dostupno svima. Korisnik koji pravi instancu klase moći će da pristupi samo atributima u javnom interfejsu.

Privatan pristup

Pristup atributima navedenim iza rezervisane reči *private* isključivo je privatan ili su atributi skriveni od korisnika klase. Privatni interfejs se naziva i detalji implementacije. Detalji implementacije opisuju kako se klasa osposobljava da radi. Uzmimo, na primer, bicikl: pedale i ručica menjača brzine su delovi javnog interfejsa. Kada pokrećete pedale, to kretanje se prenosi na točak preko zupčanika i lanca; kada pritisnete ručicu menjača brzine, mehanizam ovog ure?aja menja stepene prenosa. Zaista bi bilo "čarobno" kada biste se, dok upravljate i okrećete pedale, svaki čas saginjali da rukom premestite lanac sa jednog na drugi zupčanik. Pedale i ručica menjača predstavljaju javni interfejs, a zupčanici, lanac i mehanizam menjača - privatni interfejs.

Kada govorimo o skrivanju informacija, mislimo na privatne atribute klasa koji su skriveni od korisnika. Korisnik vašu klasu vidi kroz njen javni interfejs; privatni interfejs omogućava da ona i radi.

Prigovor objektno orijentisanom programiranju se odnosi na to što svi podaci treba da budu smešteni u privatni interfejs, i što pristup treba da im se obezbedi samo kroz svojstva. (Pogledajte odeljak "Dodavanje svojstava klasama".)

Zaštićen pristup

Preko instance objekta ne možete da pristupite njegovim zaštićenim metodama, podacima ili svojstvima. Me?utim, ako definišete novu potklasu, iz nje direktno možete da pristupite navedenim zaštićenim članovima. Podatke i metode stavite u zaštićeno područje klase kada korisnicima klase želite da ograničite pristup tim članovima, a istovremeno želite da programeri imaju mogućnost da prošire ponašanje zaštićenih metoda ili da pristupe podacima.

Publikovan pristup

Publikovan pristup ima smisla samo kod alatki za brzo programiranje. On je sličan javnom pristupu, osim što se koristi za atribute koji treba da budu dostupni u fazi projektovanja. Publikovana svojstva i atributi doga?aja

namenjeni su za korišćenje sa klasama komponenata. Publikovani atributi su oni atributi koji se prikazuju u Object Inspectoru. Sledeći kôd ilustruje sva četiri specifikatora pristupa:

```
TDemo = class
private
FSomeIntData : Integer;
protected
Procedure OverrideMe;
public
Procedure EveryOne;
published
Property
SomeIntData : Integer read FSomeIntData;
end;
```

U gornjem primeru, korisnici instance klase *TDemo* mogu da pozivaju samo njenu proceduru *EveryOne* i njeno svojstvo *SomeIntData* . Kôd sadržan u klasi direktno može da menja podatke *FSomeIntData* i proceduru *OverrideMe* . Kada se klasa implementira, neophodno je da se obrati pažnja na sve njene aspekte. Me?utim, ako neke atribute smestite u privatno ili zaštićeno područje klase, korisnici ne treba da brinu o tim atributima kada koriste instancu klase. Kada su odre?ene metode i podaci nedostupni korisniku, znatno se smanjuje napor potreban da se nauči korišćenje klase, a to upravo želimo da postignemo. (U odeljku "Dodavanje svojstava klasama" potražite objašnjenje deklaracije svojstva u programu *TDemo* .)

Automatizovan pristup

Automatizovanim identifikatorima se može pristupati kao i javnim identifikatorima, ali se u načelu koriste samo za potklase klase *TOleAuto* , a postoje samo radi kompatibilnosti sa starijim verzijama.

Unapre?ivanje i degradiranje stepena pristupa

Pristup identifikatoru pri pravljenju potklasa može se proširiti, ali se ne može suziti. Na primer, zaštićenu metodu možete u potklasi da pretvorite u javnu, ali u istoj potklasi ona ne može da postane privatna.

Izuzeci od pravila pristupanja

Ako klasu koristite unutar programske jedinice u kojoj je definisana, onda su pravila pristupa klasi manje stroga. Privatnim i zaštićenim članovima klase tada se može pristupiti potpuno javno unutar iste programske jedinice. Nezvanično objašnjenje ovakvog ponašanja glasi: samo autor klase može da do?e na ideju da pravi njenu instancu unutar programske jedinice u kojoj je klasa definisana, a on sigurno neće zloupotrebiti privatne i zaštićene atribute.

Ova mogućnost da se u Object Pascalu ne pridržavate pravila pristupanja pruža vam priliku da imitirate klase. Razmotrite slučaj kada treba da pristupite samo prevedenoj programskoj jedinici. Otkrićete da nekom važnom atributu ne možete direktno da pristupite. Ako u programskoj jedinici u kojoj želite da

upotrebite zaštićene podatke obezbedite imitaciju potklase, stvarni tip možete da pretvorite u imitirajući tip ili da napravite instancu imitirajućeg tipa i da zaštićenim atributima direktno pristupite. To je ilustrovano sledećim kodom:

```
type
TFudgeControl = class(TControl);
Procedure TForm1.ToggleState;
const
STATE_COLORS : array[Boolean] of TColor = (clGray, clWhite);
var
I : Integer;
begin
for I := 0 to ControlCount - 1 do
begin
Controls[I].Enabled := Not Controls[I].Enabled;
TFudgeControl(Controls[I]).Color :=
STATE_COLORS[ Controls[I].Enabled ];
end;
Button1.Enabled := True; // ovime se izbegava if u petlji
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
ToggleState;
end;
```

U gornjem primeru procedura *ToggleState* izvrće aktivno stanje (engl. enabled) svakog objekta kontrole, koristi logičke vrednosti stanja kao indeks niza boja i objektu kontrole dodeljuje ispravno stanje boje. Me?utim, svojstvo *Color* zaštićeno je u klasi *TControl* . Ako klasu *TControl* imitiramo klasom *TFudgeControl* , svojstvo *Color* postaće dostupno. U alternativnoj implementaciji, umesto da iskoristimo listu Controls, bilo bi potrebno da izričito navedemo sve kontrolne objekte na obrascu u proceduri *ToggleState* , a to ne bi baš bilo napredno programiranje.

Opseg

Izrazom opseg važenja (engl. scope) označavamo mogućnosti korišćenja datih atributa. Opseg se odnosi i na mogućnost korišćenja bilo kog dela koda. Kada atribut ima privatan opseg, onda se on može koristiti unutar klase u kojoj je definisan. U prethodnom odeljku opisali smo privatan, zaštićen, javan i publikovan opseg.

Treba pomenuti još neke vrste opsega: lokalni, globalni i proceduralni opseg. Globalni opseg ima sve što je definisano ili deklarisano u odeljku programske jedinice predvi?enom za interfejs. Znači, ako neki atribut ima privatan opseg, a njegova klasa je definisana u odeljku sa interfejsom, tada klasa ima globalan opseg. Identifikatorima sa globalnim opsegom važenja može da pristupi svaka programska jedinica koja koristi klasu. Postavite programsku jedinicu Unit2 u

odredbu *Uses* programske jedinice Unit1 i identifikatori interfejsa - sve što je definisano ili deklarisano u odeljku sa interfejsom - biće dostupni jedinici Unit1.

Lokalni opseg imaju svi identifikatori u odeljku sa implementacijom. U tom odeljku mogu da budu definisane i klase koje tada imaju lokalan opseg važenja. Opseg važenja, me?utim, ne ograničava se na klase, interfejs ili odeljak sa implementacijom; on se tiče mogućnosti korišćenja. Svi identifikatori imaju neku mogućnost da budu upotrebljeni. Kada se promenljiva deklariše unutar funkcije ili procedure, ona ima proceduralni opseg važenja, a to znači da se može koristiti samo unutar procedure.

U Object Pascalu možete da definišete ugnež?ene procedure. Takva procedura se definiše iza naredbe za deklarisanje procedure, a ispred bloka *begin/end* . Ugnež?ena procedura se može pozivati samo unutar procedure u kojoj je definisana.

```
procedure Proc;
procedure DoSomething;
begin
ShowMessage('Ja sam ugnezđena.');
end;
begin
DoSomething;
end;
```

Promenljive definisane unutar bloka procedure mogu se koristiti samo u toj proceduri ili u ugnež?enim procedurama; definicije u proceduri imaju proceduralni opseg važenja, što je uže od opsega implementacije. Promenljive ili procedure definisane izvan procedure, ali unutar odeljka sa implementacijom, mogu se koristiti samo unutar tog odeljka. Sve što je definisano u odeljku sa interfejsom može se koristiti bilo gde u programskoj jedinici, a može ga koristiti i bilo koji kôd koji koristi programsku jedinicu. Promenljive u odeljku sa interfejsom imaju globalan, najširi, opseg važenja. Kada shvatite pravila za opseg važenja, umećete da prikrijete informacije, odnosno da ograničite pristupanje promenljivama i procedurama tako što ćete ih učiniti dostupnim samo delovima programa koji su neophodni za njihovo ispravno i efikasno korišćenje. Kada se koristi globalni opseg važenja, kao u interfejsu, možete da ograničite pristup odre?enim podacima i procedurama ako ih smestite u klasu, a zatim ograničite pristup klasi odgovarajućim specifikatorima.

Cilj prikrivanja informacija

Pravila pristupanja informacijama nisu smišljena da pomognu računarima, nego ljudima. Njihov cilj je jasan: ako informacije uspemo da raspodelimo u što manji broj jasno odeljenih grupa, ljudski um će moći da rešava složene probleme. Što je veća mogućnost grupisanja informacija, problem postaje jednostavniji.

U sledećim primerima objasnićemo najbolje načine za skrivanje informacija koji provereno pomažu da se smanji složenost.

1. Za podatke i procedure upotrebite što uži opseg važenja, ograničavajući im dostupnost na neophodnu meru. Na primer, ako je funkciji potrebna promenljiva, definišite promenljivu u oblasti važenja funkcije, a ne u širem opsegu gde postoji mogućnost da se njena vrednost slučajno promeni.
2. Pristupanje elementima klasa ograničite što je više moguće. Svaki član treba da je privatan, osim ako postoji jak razlog da bude dostupniji.
3. Podaci uvek treba da su privatni; pristupanje podacima obezbedite preko svojstava, što samo po sebi omogućuje da ga suzite.
4. Ako identifikatore smeštate u odeljak sa interfejsom programske jedinice, neka oni po mogućnosti budu unutar klase.

Imajte na umu da je računaru svejedno gde smeštate informacije. Što se njega tiče, sve identifikatore možete da smestite u odeljak sa interfejsom, a sve atribute klasa u odeljak sa javnim pristupom. To će vam na početku olakšati posao, ali pomalo liči na poklonjenu kreditnu karticu - na kraju ćete sve morati da platite.

Dodavanje svojstava klasama

Svojstva su novina u Delphiju. U načelu, i svojstva su samo podaci, ali pošto nude i nove mogućnosti, to su "pametni" podaci. Budući da pojava Delphija znači konačan razlaz sa staromodnim stilom integrisanog okruženja za programiranje, svojstva su uvedena da bi odgovorila dodatnim zahtevima nove alatke za brzo programiranje.

Podaci održavaju stanje objekta

Tokom niza godina u industriji softvera bilo je dobro poznato da su globalne vrednosti loše rešenje. Objektno orijentisanim programiranjem uveden je pojam kapsuliranja podataka u klasama da bi se njima moglo bolje upravljati. Nažalost, modifikovanje podataka pomoću procedura pokazalo se složenim jer se pomoću procedura teško upravlja podacima; na primer, u jednostavnom izrazu ne možete kao operand da upotrebite proceduru. Pošto je takva upotreba procedura bila složena, nije dosledno ni primenjivana. Ako se procedure ne koriste za upravljanje podacima, onda su podaci u klasama izloženi zloupotrebi kao i globalni podaci. Povrh toga, procedure se nisu mogle pozivati u integrisanom okruženju za programiranje u toku projektovanja; njihovi vizuelni atributi potrebni za oblikovanje grafičkog okruženja nisu se mogli dinamički modifikovati.

Nabrojani problemi su ukazivali da objektno orijentisanom svetu nešto nedostaje: "pametni" podaci. Već dugo je u industriji softvera poznato da je pristupanje podacima preko funkcija efikasna tehnika za zaštitu podataka od zloupotrebe. Problem je bio kako podatke vezati za funkcije provere, a da se oni i dalje ponašaju kao podaci. Odgovor je na?en u svojstvima, relativno novom pojmu u objektno orijentisanom programiranju.

Svojstva omogućavaju pristupanje podacima

Svojstva predstavljaju podatke. Pri definisanju svojstva koristite naročitu sintaksu koja svojstvo vezuje za vrednosti odre?enih podataka. Evo najjednostavnije definicije svojstva:

property imepromenljive : tippodataka read imepolja write imepolja

Reči *property* , *read* i *write* u definiciji se pišu doslovno. *Imepromenljive* vi zadajete. Kao i kada zadajete ime promenljive, treba da upotrebite cele reči, najbolje imenice. *Tippodataka* je bilo koji tip koji se može primeniti na promenljivu, uključujući ugra?ene tipove, klase ili nabrojane tipove, ili proceduralne tipove. *Imepolja* u odredbi *read/write* predstavlja stvarnu vrednost podatka. Neophodno je da *imepolja* bude atribut podataka klase istog tipa kao i svojstvo; u krajnoj liniji, *imepolja* jeste element podataka. Vrednost polja je po konvenciji privatna i dobija prefiks *F* , a ime svojstva je identično imenu polja, samo bez *F* . Na taj način se polja i svojstva lako sparuju, kao u sledećem primeru.

```
TDog = class
private
FColor : TDogColor;
FGender : TDogGender;
FBreed : TDogBreed;
FWeight : TDogWeight;
public
Procedure Jump;
Procedure Run;
Procedure Bark;
Procedure Sleep;
property Color : TDogColor read FColor write FColor;
property Gender : TDogGender read FGender write FGender;
property Breed : TDogBreed read FBreed write FBreed;
property Weight : TDogWeight read FWeight write FWeight;
end;
var Dog : TDog;
begin
Dog := TDog.Create;
Dog.Color := clYellow; // Stari Æuća
// ...
end;
```

Svojstva se ne mogu proslediti procedurama kao argument *var* , niti se može upotrebiti adresa svojstva.

Preklopljeni operatori u jeziku C++ omogućuju implicitno pozivanje funkcija kada se pristupi podacima, mada je sintaksa mnogo komplikovanija i stoga podložna greškama.

Obratite pažnju na to da su broj i raznolikost polja u klasi *TDog* identični broju i tipovima svojstava. To nije neophodan zahtev i ako samo možete da udružite vrednosti podatka sa svojstvom, onda polja sama po sebi ne donose veliku korist. U stvari, to nije tako jednostavno. Odredba *read/write* može da ima funkciju za *read* , a proceduru za *write* . To znači da se ono sa čime postupamo gotovo kao sa prostim tipovima podataka u stvari može filtrirati kroz metodu.

Svojstva rešavaju problem održavanja jednostavnosti podataka omogućujući istovremeno kontrolisano korišćenje privatnih podataka kojima se pristupa pomoću javnih metoda. Osim odredbi *read* i *write* , postoje i složenije tehnike koje ne možete da primenite na sirove podatke, uključujući i definisanje svojstava samo za čitanje (read--only), odnosno samo za pisanje (write-only). (Pogledajte poglavlje 8.) U nastavku objašnjavamo implementaciju metoda *read* i *write* .

Metode za pristupanje svojstvima

Svojstvo čitanja (read) uvek odgovara tipu podataka na koje se odnosi ili predstavlja funkciju koja vraća vrednost tipa identičnog svojstvu. Metoda upisivanja (write) je ili referenca na polje tipa istog kao i svojstvo ili je procedura koja prihvata argument tipa jednakog tipu podataka.

Metode read

Kod jednostavnih svojstava, metode *read* obično imaju oblik: *function Get imepolja* : *tippodataka* ;. *Get* je glagol koji se koristi po konvenciji. Metode *read* se pozivaju implicitno kada se svojstvo upotrebi kao vrednost sa desne strane, tzv. dvrednost. Pošto nije predvi?eno da ih korisnici izričito pozivaju, metode read se obično smeštaju u odeljak klase sa zaštićenim ili privatnim pristupom. Nastavljajući prethodni primer sa klasom *TDog* , dodajmo metodu *read* svojstvu *Color* :

```
protected
Function GetColor : TDogColor;
public
Property Color : TDogColor read GetColor write FColor;
...
implementation
Function TDog.GetColor : TDogColor;
begin
result := FDogColor;
end;
```

Navedeni deo koda treba kao modifikaciju ugraditi u prvobitni kôd klase *TDog* . U ovom primeru se vidi da je metoda svojstva *GetColor* propisno implementirana u odeljku sa implementacijom. Sa instanciranjem klase *TDog* bila bi pozvana metoda *read* za svojstvo *Color* ako bi to svojstvo bilo upotrebljeno kao dvrednost, npr. ovako:

```
Dog := TDog.Create;
if( dcBrown = Dog.Color ) then
// dalji kôd
```

U gornjem listingu bi metoda *read* bila pozvana i ako bi iskaz *Dog.Color* bio na levoj strani operatora = zato što poziv samo procenjuje podatke, ne menjajući ih.

Metode write

Jednostavna metoda *write* ima oblik: *Procedure Setimepolja(const Value : tippodatka)* ; gde se prefiks *Set* koristi po konvenciji kao glagol koji označava akciju a *imepolja* je imenica koja predstavlja predmet nad kojim se vrši akcija. Nastavljamo sa primerom klase *TDog* dodajući joj dalje modifikacije:

```
protected
Function GetColor : TDogColor;
Procedure SetColor( const Value : TDogColor );
public
Property Color : TDogColor read GetColor write SetColor;
...
implementation
Procedure TDog.SetColor( const Value : TDogColor );
begin
if( Value = FColor ) then exit;
FColor := Value;
end;
```

U primeru je upotrebljen specifikator *const* jer je argument nepromenljiv, a to je odlika konstanti. U ovom primeru programski red *if(Value = FColor) then exit* ; sprečava ažuriranje vrednosti podatka ukoliko je ona već ispravna. To je odlična tehnika koja se može primeniti na baze podataka gde treba izbeći nepotrebno i skupo ažuriranje baze ili grafičkog okruženja jer to može neprimereno da zaposli procesor.

Sažetak

U poglavlju 2 obuhvaćeno je sve ono što omogućava da Windows sara?uje sa programom Delphi i da Delphi sara?uje sa Windowsom. Windows je operativni sistem zasnovan na porukama koje izazivaju doga?aji. Delphi je izgra?en na Object Pascalu, objektno orijentisanom programskom jeziku koji se prirodno preslikava u Windowsove poruke i doga?aje. Klase, metode, podaci i svojstava osnovni su gradivni blokovi svake aplikacije napravljene u Delphiju.