BLOG

KOMPLETAN VODIČ

BIBLIOTEKA *sve tajne*

Luka Abrus i Domagoj Pavlešić

Microsoft .NET

simfonija programiranja

Postati programer nikad nije bilo jednostavnije. Za ostvarenje tog cilja potrebne su vam samo dvije stvari: Microsoftova platforma .NET koja je svojom pojavom dubinski izmijenila i pojednostavila život programera, te ova knjiga, ultimativni vodič kroz sve tajne te platforme. Bez obzira na to jeste li programer početnik ili iskusni koder, knjiga će vas pitkim štivom prošetati kroz arhitekturu, osnove programiranja i razvojnu okolinu te ćete uz idealan omjer teorije i konkretnih primjera naučiti stvarati prozorske i web-aplikacije, uključujući i web-servise.

UKLJUČUJE DVD S *TRIAL* INSTALACIJOM Visual Studio .NET Professional

Domagoj Pavlešić i Luka Abrus

MICROSOFT .NET SIMFONIJA PROGRAMIRANJA

BUG & SysPrint

Zagreb, 2004.

Domagoj Pavlešić i Luka Abrus

MICROSOFT .NET SIMFONIJA PROGRAMIRANJA

Izdavači:
Bug d.o.o. za novinsko-nakladničku djelatnost
SysPrint d.o.o za nakladničku djelatnost

Urednik:
Dragan Petric

Urednički kolegij:
Dragan Petric
Miroslav Rosandić
Robert Šipek

Lektura i korektura:
Branka Savić

Grafički dizajn:
Ivana Miličić

Naslovnica:
Tomislav Stanojević

Priprema za tisak:
Vesna Milak
Tomislav Stanojević

Tisak:
Tiskara "Printel", Zagreb

ISBN 953-6717-19-0 (Bug)
ISBN 953-232-049-0 (SysPrint)

CIP – Katalogizacija u publikaciji
Nacionalna i sveučilišna knjižnica - Zagreb

UDK 004.4 .NET (035)

ABRUS, Luka

Microsoft .NET : simfonija programiranja / Luka Abrus i Domagoj Pavlešić. - Zagreb : Bug : Sysprint, 2004. - (Biblioteka Sve tajne)

Kazalo.

ISBN 953-6717-19-0 (Bug). -
ISBN 953-232-049-0 (SysPrint)

1. Pavlešić, Domagoj
I. Kompjutorsko programiranje - - .NET tehnologija

440330174

Domagoj Pavlešić
Luka Abrus

Microsoft .NET

simfonija programiranja

O autorima

Domagoj Pavlešić, rođen je 27. svibnja 1980. godine u Zagrebu. Tu je pohađao osnovnu školu, matematičku gimnaziju (legendarni MIOC) te zatim upisao Ekonomski fakultet gdje na smjeru marketinga polako, ali uporno zarađuje titulu dipl. oec.

Prvo računalo bilo mu je ZX Spectrum kojem je ugrađeni kasetofon bio potrgan. U silnoj želji da što prije isproba novog kućnog ljubimca nabavio je knjigu o programiranju, a kada je kasetofon par tjedana kasnije bio popravljen, više za njega nije htio ni čuti.

Još od kraja osnovne škole predstavljao je svoje softverske uratke na smotrama mladih informatičara, a prvu "komercijalnu" aplikaciju napisao je u tada superpopularnom Clipperu. Uskoro prelazi na Windowse gdje se igra s Delphijem, no svojom ga pojavom Internet toliko zaluđuje da se predaje izradi web-aplikacija.

U prilično bogatoj karijeri *webmastera* napravio je više desetaka web-stranica od kojih mu je i dalje najdraži Bug On Line za koji se brine posljednjih pet godina. Usto, kao voditelj web-studija Dizzy internetska rješenja posredno i neposredno svakodnevno radi na barem nekoliko internetskih projekata.

U svojoj spisateljskoj karijeri pisao je u brojnim informatičkim časopisima i novinama, sve dok se 1998. godine, na poziv glavnog urednika, nije uvukao u Bug među najplodnije stalne suradnike gdje je zadovoljan ostao do današnjih dana. U međuvremenu se počela izdavati i Mreža koja mu je po izboru tema nešto bliskija pa veći dio svojih tekstova objavljuje u tom izdanju. Nedavno su ga pokušali navući i na pisanje za Enter, no što je previše, previše je.

Voli jezike i silno bi želio naučiti portugalski. Voli putovanja i nada se putu u Južnu Ameriku. Obožava putovati svim prijevoznim sredstvima, no najveće veselje predstavlja mu let avionom. Želi letjeti balonom. Lud je za automobilima, jako voli voziti i biti vožen. Zaljubio se u skijanje, neobjašnjivo mu se sviđa kuglanje, uživa u vodi. Voli arhitekturu i kvalitetnu glazbu. Pjeva u zboru. Ne pije kavu. Obožava planine u svim godišnjim dobima. Domoljub je i nikada ne bi živio u inozemstvu.

Luka Abrus, rođen je 7. rujna 1981. godine u Zagrebu. Apsolvent je računarstva na FER-u, kamo je došao poslije matematičke gimnazije. Od ostalog obrazovanja ima završenu srednju glazbenu školu te nekoliko tečajeva stranih jezika.

Svoje početke veže uz izradu web-stranica – prvu je napravio početkom 1997. godine, a otada je u sklopu web-studija Dizzy internetska rješenja izradio više desetaka *siteova*. (Zapravo, ako ćemo biti potpuno iskreni, svoje početke veže uz igranje igara na Commodoreu 64, no to ne zvuči kao nešto za biografiju.)

Svi poslovi koji su mu donijeli neki prihod okreću se oko računala. Od rujna 2003. godine radi kao .NET Evangelist u Microsoftu Hrvatska, što mu daje i službeno pravo da bude zadivljen .NET tehnologijom i da širi "dobru vijest" među pukom. No istinski se zaljubio u .NET na prijašnjim poslovima gdje je radio kao programer web-aplikacija.

Višegodišnji je stalni suradnik časopisa Bug i Mreža u kojima je i stekao iskustvo u pisanju radeći na temama usko vezanim uz Web i programiranje. Broj znakova koje je objavio u tim časopisima se polako, ali sigurno, približava milijunu i pol.

Ovo mu je treća objavljena knjiga. Dosad je u biblioteci "Sve tajne" objavio uspješnicu pod nazivom *Izrada Weba – abeceda za webmastere* koja pokriva apsolutno sve vezano uz izradu web-stranica, a u biblioteci "Brzi vodič" objavio je knjigu o izradi web-stranica u Dreamweaveru, znakovitog naziva *Brzi vodič kroz Dreamweaver MX 2004*.

Nakon programiranja i pisanja, najvažnija stvar u životu mu je glazba, a svirajući razne instrumente u prstima je sakupio poveći broj svirki. Sluša glazbu kad god stigne i voli pratiti sve sportove. Katkad i sam zaigra košarku, obožava skijanje, a ni plivanje i vaterpolo mu nisu strani.

U životu želi nastaviti raditi što i sad radi, naučiti svirati još koji instrument, malim brodom oploviti neko more, svirati u dobrim klubovima, izmisliti nešto novo i putovati svijetom. U životu ne želi raditi neplaćene poslove, biti vođa neke navijačke skupine, služiti kao negativan primjer te sjediti doma i debljati se.

Zahvale

Napisati ovakvu gomilu teksta izuzetan je napor, čak i kada je cijeli posao podijeljen na dvije osobe. Fizički i psihički simptomi koji se javljaju tijekom tog razdoblja utječu, kako na autore, tako i na ljude koji se nalaze oko njih pa je red posvetiti im koju riječ zahvale.

Ipak, prije nego što počnemo nabrajati sve zaslužne za ono što jesmo i ono što smo napisali, red je da se – ma koliko to patetično zvučalo – zahvalimo jedan drugome. Suradnja ovog tipa zahtijeva veliku količinu povjerenja, strpljivosti i kompatibilnosti mišljenja, što se u našem slučaju nevjerojatno poklopilo unatoč ponekim natezanjima oko, iz perspektive cijele knjige, nevažnih sitnica.

Slijedi zahvala kompletnoj ekipi u Bugu, počevši od kompletne postave urednika koji su nam svojim dugogodišnjim zadavanjem i odobravanjem tema istesali spisateljske sposobnosti. Bez zaista bogatog iskustva pisanja za Bug, Mrežu i ponekad Enter napisati tekst ovakvih razmjera bilo bi jednostavno nemoguće. Miro, Drago, Oleže – hvala vam!

Treba zahvaliti i uredniku Bugovih knjiških izdanja, Draganu Petricu i njegovu šefu iz sjene, Robertu Šipeku, što su nas natjerali da se primimo pisanja ove knjige ma koliko nam se to u početku činilo (pre)velikim zalogajem. Zapravo, još uvijek mislimo da se radi o velikom zalogaju, no to samo povećava radost da su dani i noći pisanja iza nas...

Veliko hvala upućujemo i Kristini Lovrić, koja je često na prepad presuđivala u nevjerojatno burnim raspravama oko jezičnih dilema i stručno birala strane između Domagojevih “ovo-je-pravilnije” i Lukinih “slažem-se-ali-ovo-je-bolje” stavova.

Zahvalnost dugujemo i Ratku Mutavdžiću (Microsoft Consulting Services), jednoj od najvećih faca u hrvatskoj *developerskoj* zajednici, a i šire, koji se rado prihvatio pisanja predgovora za ovo djelo.

Velika nam je čast što je predgovor napisala tako kompetentna osoba s dubokim tehnološkim *backgroundom* – Ratko, hvala!

Slijedi zahvala i cijeloj ekipi iz Microsofta Hrvatska na čestim pitanjima o statusu knjige, što nam je dalo veliki poticaj da sve ovo napišemo, a ujedno i predstavljalo obavezu zbog njihovih očekivanja i pohvala. Posebnu zahvalu zaslužuju Tihomir Cirkvenčić i Bojan Hadžisejdić zbog velikog truda i pomoći da se uz ovu knjigu pojavi i DVD.

Iako je to pomalo neobično, želimo se zahvaliti i Microsoftu kao instituciji zbog njihove velike brige o *developerima*, a posebno na postojanju tako detaljne, bogate i opsežne dokumentacije bez koje bi pisanje ovakve knjige bilo, bez imalo pretjerivanja, barem trostruko teže. Blagodati tog odnosa, vjerujemo, osjetit ćeš i ti, dragi čitatelju, kada se otisneš u vode koje ne pokriva ova knjiga.

Od srca hvala i svim prijateljima i prijateljicama koji su nas bodrili tijekom pisanja, raspitivali se o napretku i obzirno nenagovarali na izlaske kako bismo uhvatili postavljene rokove. Hvala za druženja, razgovore, događanja i inicijative, posebno stoga što nisu bili vezani uz računala te su tako predstavljali nužno potreban mentalni odmor. Tko zna bismo li se bez vas uopće odljepljivali od tipkovnice...

I konačno, kao šećer na kraju, najveće hvala našim obiteljima – na podršci, razumijevanju i pažnji kojima smo napajani ne samo tijekom pisanja knjige nego tijekom cijelog života. Bez podrške obitelji teško je išta postići, a mi smo, što se toga tiče, zaista imali sreće. Hvala!

Pregled sadržaja

Predgovor

Rijetke su (i sretne) generacije koje mogu bili dijelom revolucije. Naravno, ovdje ne mislim na revolucije koje su provedene mačem i ognjem, uz čudnovata konačna rješenja zbog kojih su iznova izbijale nove revolucije i evolucije. Tehnološke revolucije ne događaju se često: u zadnjih nekoliko tisuća godina, čovječanstvo ih je doživjelo tek nekoliko – od industrijske krajem pretprošlog stoljeća do informatičke, koja se još uvijek odvija (iako neki moderni mislitelji misle da je završila davnih dana i da je sve ovo što se događa oko nas tek n-ta iteracija istih događanja).

Kao i društvo, tako i organizacije imaju svoje revolucije. Ponekad su uspješne u njima, a još češće se događa da ih upravo revolucija koja se odvija u njihovu okruženju ostavi zbunjena pogleda koji rezultira konačnim krahom organizacije (što bi rekli naši ljudi, ključ u bravu). Za razliku početnih revolucija koje su se događale svakih nekoliko stotina godina, informatičke revolucije se događaju na godišnjem nivou – upravo se svake godine po nekoliko novih tehnologija predstavi tržištu kao *breakthrough* tehnologija koja će okrenuti i pokrenuti svjetsko tržište. Neke od istih u tome zbilja i uspiju, dok se većina tragom i netragom izgubi već u sljedećih nekoliko mjeseci.

Zgodno je napomenuti da se, za razliku od nekih drugih industrija, tehnologije u informatičkom svijetu uvijek vraćaju kao bumerang – nešto što smo mislili da je netragom propalo jer je bilo neprimjenjivo ili preskupo pojavljuje se kao ponovni pokušaj već u sljedećem uzletu tržišta, kao "ovaj put će to sigurno ići". Sjetite se samo Appleova Newtona i Microsoftova PenPC računala i usporedite to s današnjom Microsoft Tablet PC tehnologijom. Međutim, puno su važnije revolucije koje mogu pokrenuti ili uništiti neku organizaciju. Ako ste imali prilike pročitati odličnu knjigu Claytona M. Christensena *Innovator's Dilemma*, onda znate o čemu pričam: pojedine informatičke revolucije bile su toliko silne i brze da se velike, dobro uhodane organizacije nisu uspjele niti okrenuti za brzim trkačem koji je prohujao kraj njih. Dobar primjer dat je u proizvodnji tvrdih (hard) diskova za računala: tko se danas još sjeća Conner, Seagate, Miniscribe, Quantum, Memorex ili Shugart proizvođača tvrdih diskova, koji su u svoje respektivno vrijeme bili upravo broj jedan na tržištu. Ovaj potonji (Shugart) čak je i izumio tvrdi disk!

U tim vremenima revolucije su se događale svake dvije-tri godine: pojavljivali su se novi oblici tvrdih diskova: 8, 5.25, 3.5, 1.8 inča tvrdi diskovi izbacivali su se zajedno sa smanjivanjem i promjenom namjene osobnih računala. Organizacije koje su u to vrijeme vodile tržišnu bitku imale su i više nego dovoljno posla s isporukama diskova za više nego rastuće tržište osobnih računala – dok su organizacije koje su tek startale s proizvodnjom imale dovoljno vremena posvetiti se malim serijama novih tipova diskova za nove tipove računala (recimo, prijenosna računala). I upravo je nedostatak trenutnog tržišta za nove tehnologije uništio moćne, dobro pozicionirane organizacije – kada su shvatile da nastaje novo tržište, već je bilo prekasno – već su postojali novi lideri, i stari su jednostavno nestali.

No čemu ovakav uvod kad pišem predgovor za knjigu o programskoj podršci (čitaj: softver)? Microsoft je tek jedan o tehnoloških i tržišnih lidera na informatičkom tržištu. Uvijek me iznova zabavlja činjenica (s obzirom na percepciju koju većina čitateljstva i gledateljstva ima o ovoj tvrtki) da Microsoft drži tek 3% ukupnog svjetskog informatičkog tržišta (no, naravno, u pojedinom segmentima, poput Microsoft Officea, drži i mnogo više). Iako ga većina ljudi drži informatičkim monopolistom, Microsoft još uvijek vrlo aktivno traži svoje inovativne tehnologije i bori se za poziciju na tržištu – jedna od takvih inovacija sigurno je i .NET Framework niz tehnologija.

Za Microsoft je to jedan od evolutivnih koraka koji donosi bolju, kvalitetniju, bržu i sigurniju razvojnu platformu budućnosti. Za developere je to put u razvojnu budućnost koji donosi nove mogućnosti razvoja Windows i Internet zasnovanih aplikacija na Microsoft platformi. Za Microsoft je to ujedno i budućnost kompanije – svi novi sustavi, poslužitelji i rješenja temelje se na .NET Framework tehnologiji, dok će se nadolazeći operativni sustav Microsoft budućnosti pod kodnim imenom Longhorn (o kojem u trenutku pisanja ovog predgovora svi već sve znaju) u potpunosti temeljiti na .NET Frameworku. Odnosno, da pojednostavim izlaganje, budući operativni sustav bit će .NET Framework. Znači li to da bi ga bilo dobro poznavati što bolje? Naravno – time i ova knjiga koja je odličan uvod u Microsoft .NET svijet za *developere*.

Što ćete naći u ovoj knjizi? Kao što već sam naslov kaže, sadržaj objašnjava Microsoft .NET Framework – glavnu Microsoftovu investiciju u budućnost organizacije. Ali ako tražite lagani uvod u .NET Framework filozofiju, onda ćete pročitati tek nekoliko početnih poglavlja (dva!). Ostatak knjige je, naravno, namijenjen programerima, tako da je i sadržaj prilagođen upravo ovoj publici te ćete pronaći dosta izvornog koda, primjera iz prakse te raznoraznih savjeta i trikova koji će bilo kojem programeru, početniku i onome tko se već okušao u morima .NET programiranja, svakako pomoći da bolje razumije platformu, ali i da se otisne u dublje poznavanje razvojne okoline i njenih specifičnosti.

Najbolje je knjigu pročitati uz istodoban rad na računalu kako bi se znanja i koncepti odmah mogli provjeriti i u praksi. Autori na to i računaju: iako je, kao i u svakoj dobroj knjizi, na početku ostavljeno dovoljno prostora za upoznavanje s osnovama .NET Frameworka, prvi izvorni kôd možete naći već u drugom poglavlju knjige i upravo vas zove na isprobavanje na postavljenoj inačici Microsoft Visual Studija. Kako su autori ujedno i vrsni programeri s podosta iskustva, pronaći ćete vrlo malo marketinških priča – ono što je developerima kruh i voda, a to je izvorni kôd, sastavni je dio svih poglavlja i svih odjeljaka knjige. No ne treba zaobići niti one s jačim programerskim iskustvom koji uvijek trebaju najmanje dvije stvari: referentni priručnik koji će ih podsjetiti kao napraviti ono što im treba (i to je pomalo komplicirano kad koristite složena razvojna okruženja kao što je Visual Studio .NET) ili brzi priručnik za prijelaz sa standardne "stare" Visual Studio 6.0 razvojne okoline na novu (iako u knjizi nećete naći upute kako to napraviti, ali to joj ni nije bio cilj).

Na kraju i riječ-dvije o autorima. Kad malo bolje razmislim o dobrim prošlim vremenima, nema mi druge nego ih nazvati – Mirko i Slavko. Sjećam se dobro kraja devedesetih godina, kada je drugi naraštaj informatičkih genijalaca u Hrvatskoj (dakle onaj koji je odrastao na raznoraznim ZX 81 i Spectrum, Commodore i Amiga kantama) zamijenjen mlađim naraštajima, onima koji su jednostavno preskočili POKE i PEEK generaciju te odmah prionuli na Internet i slične novotarije od kojih se dijelu prethodne generacije dizala kosa (ona koja je preostala) na glavi.

Novi val bilo je lako prepoznati kako na različitim informatičkim događanjima tako i po uporabi novog digitalnog medija – Interneta; za iskazivanje kreativnosti i drugačijeg poimanja informatike, recimo da bi je trebali nazvati zabavnom. Naši junaci su imali sreće da spadaju negdje između – dovoljno su stari da se s tugom prisjećaju POKE i PEEK programiranja, a opet dovoljno mladi da odmah sa žarom uskoče na novi val Interneta. Počeli su se pojavljivati članci u starom analognom mediju (čitaj: časopisima) koji su objašnjavali kako i na koji način iskoristiti prednosti novog, neograničenog prostora: počelo je s osnovama Interneta, kako napraviti svoju prvu HTML stranicu, zatim kako napraviti web-aplikaciju, a još uvijek nije završilo (zanimljivo je pratiti Luku i Domagoja kako već nekoliko godina neumorno objašnjavaju kako napraviti neku JavaScript funkciju, kako koristiti XML ili kako pristupati bazama podataka iz aplikacija). Ali, Bože moj, uvijek ima početnika.

Grupirani za početak unutar Dizzy grupacije, ova grupa, a isto tako i web-site (postoje još i danas, samo su se dečki u međuvremenu profesionalizirali i zaključili da ipak treba zaraditi neke pare na stečenom znanju i postojećim idejama), već se u startu odlikovala svježim idejama, kreativnošću i izrazitim naprednim znanjem web-tehnologija (pažljiviji pratitelj domaće informatičke scene zna da su cijelo vrijeme kao vođe pokreta figurirale zapravo tri osobe – ali s obzirom na to da su samo dvije od njih napisale ovu knjigu, uputit ćemo samo mali pozdrav trećoj). Budući da su ujedno pisali za eminentne domaće časopise kao što su Bug i Mreža, hrvatsko tržište postalo je bogatije za još nekoliko mladih, energičnih i nadasve inventivnih ljudi, koji danas, eto, prenose svoje već bogato znanje na nove (ali i stare) naraštaje.

Ratko Mutavdžić
Engagement Manager
Microsoft Consulting Services

Uvod

Biti programer u današnjem svijetu je teško, a to tek postati još teže. Svakodnevno vas bombardiraju desecima različitih tehnologija, programskih jezika, metoda programiranja, što nužno dovodi do zasićenja i primisli da je programiranje bolje ostaviti *nekom drugom*. No dolaskom .NET-a sve se to promijenilo – .NET predstavlja revolucionarnu i već dokazanu tehnologiju koja unificira programske modele i čini razvoj web-aplikacija gotovo identičnim razvoju prozorskih aplikacija ili pak razvoju aplikacija za mobilne uređaje. Ne tako davno niste se mogli baviti tako širokim spektrom i bila je nužna specijalizacija. Danas je dovoljno svladati .NET i možete se smatrati stručnjakom za svako od ovih područja (a i mnogo šire).

Cilj ove knjige je pokazati osnovne mogućnosti .NET-a i dati vam pregled dostupnih tehnologija. Kao što ćete i sami vidjeti kroz druženje uz ovu knjigu, nakon nje ćete biti u stanju izrađivati različite vrste aplikacija, no što je mnogo važnije – razumjet ćete pozadinsku priču i imat ćete dobre temelje za unapređivanje svog znanja.

Upoznavanje s knjigom

Knjiga je podijeljena u 4 dijela, od kojih svaki obuhvaća nekoliko poglavlja slične tematike. U nastavku ćete se brzinski upoznati s koncepcijom knjige i sadržajem pojedinih poglavlja.

Prvi dio: .NET iznutra

Prva tri poglavlja upoznat će vas s osnovnim pojmovima bez kojih se ne možete upustiti u izradu aplikacija na .NET-u.

Prvo poglavlje, **Uvod u .NET**, dat će prikaz tehnologije i njenog značaja. Ključan je i pogled u budućnost koji govori da će .NET još dugo biti Microsoftova programerska platforma.

Drugo poglavlje naslovljeno je **Arhitektura .NET-a**, a upoznat će vas s načinom rada .NET-a. Poznavanje arhitekture nužno je za razumijevanje kasnijih poglavlja knjige pa će vam to postaviti temelje za daljnje čitanje.

U trećem poglavlju **Programski jezici** ući ćete u svijet programiranja u .NET-u prikazano kroz osnove dvaju najpopularnijih jezika, C# i Visual Basic .NET. Naučit ćete osnovne koncepte programiranja uspoređujući kako se to radi u pojedinom jeziku.

Razvojna okolina je naziv četvrtog poglavlja i služi za upoznavanje s Visual Studijom .NET. Radi se o programu koji se koristi kroz čitavu knjigu i služi za izradu svih tipova aplikacija. Visual

Studio .NET glavna je razvojna okolina za .NET i poznavanje njegovih temeljnih mogućnosti osnovica su svih poglavlja.

Drugi dio: Osnove programiranja

Nakon početnog upoznavanja s .NET tehnologijom i razvojnom okolinom s kojom ćete se družiti u razvoju aplikacija, u drugom dijelu knjige ćete se upoznati s osnovama programiranja u .NET-u.

Peto poglavlje, naslovljeno **Tipovi podataka,** naučit će vas kako koristiti podatke različitih tipova u svojim aplikacijama. Susrest ćete se i s metodama pretvorbe podataka i s klasama.

Klase ćete pobliže upoznati u šestom poglavlju **Objektno orijentirano programiranje** koje će vas naučiti osnovne objektnog modela u .NET-u i koncepte objektno orijentiranog programiranja, kao što su učahurivanje, višeobličje, nasljeđivanje i mnogi drugi, a upoznat ćete i različite tipove klasa te sučelja.

Iznimke ćete upoznati u istoimenom sedmom poglavlju. Pojavljivanje neočekivanih grešaka u radu aplikacija je stvarnost te je stoga nužno poznavanje metoda upravljanja i hvatanja iznimki.

Treći dio: Dijelovi .NET-a

Prva dva dijela knjige dat će vam prijeko potrebne temelje za rad u .NET-u. Nakon što detaljno proučite svako od prethodnih poglavlja, bit ćete spremni za pravu stvar – izradu aplikacija.

Najprije ćete izrađivati u prozorske aplikacije u osmom poglavlju pod naslovom **Windows Forms**. Naučit ćete sve detalje o kontrolama, događajima u aplikacijama, radu s formama i korištenju pisača.

Deveto poglavlje nosi naziv **ADO.NET** i upoznat će vas s istoimenom tehnologijom za pristup podacima. Nakon što ga pročitate znat ćete kako izrađivati aplikacije koje koriste baze podataka te osnove jezika SQL za pristup podacima u bazama.

Deseto poglavlje, **ASP.NET**, uvest će vas u svijet programiranja web-aplikacija. Naučit ćete razvijati aplikacije na najmoćnijoj serverskoj tehnologiji, kako ih konfigurirati i optimizirati naprednim metodama te povezati s naučenim u prethodnom poglavlju, ADO.NET tehnologijom.

XML je naziv jedanaestog poglavlja i demistificirat će tehnologiju što se krije iza te kratice i kako je možete koristiti u svojim aplikacijama. Upoznat ćete i osnove XPatha, jezika za pristup podacima u XML-u, što ćete uvelike koristiti u svojem radu s XML-om.

Prava poslastica skriva se u dvanaestom poglavlju pod naslovom **Web-servisi,** u kojem ćete naučiti sve o izradi web-servisa i njihovu korištenju iz aplikacija pisanih u .NET-u.

Četvrti dio: Dodaci

Dodatak A obrađuje **.NET Compact Framework** odnosno izradu aplikacija za mobilne uređaje. Upoznat ćete posebnosti verzije .NET Frameworka prilagođene mobilnim uređajima, a tako ćete i proširiti broj platformi za koje možete izrađivati aplikacije u Visual Studiju .NET.

Posebno koristan bit će vam dodatak B naslova **Otklanjanje grešaka** jer na jednostavan način objašnjava sve metode koje vam stoje na raspolaganju u Visual Studiju .NET za *debugiranje* aplikacija. Upoznat ćete različite načine kontroliranja izvršavanja kôda, praćenja stanja varijabli i objekata te korištenja prekidnih točaka.

Dodatak C, **Pomoćni alati za .NET**, donosi vam kratak popis svih alata dostupnih pod .NET-om koji mogu olakšati izradu vaših aplikacija ili povećati kontrolu nad načinom njihova rada.

Dodatak D, **DVD**, ukratko opisuje prateći DVD koji ste dobili s knjigom, a sadrži 60-dnevnu potpuno funkcionalnu probnu verziju Visual Studija .NET 2003 Professional.

Na kraju knjige pronaći ćete i detaljan rječnik pojmova spominjanih u ovoj knjizi, direktno vezanih za programiranje u .NET-u te abecedno kazalo.

Što ćete naći u knjizi?

Pred sobom imate knjigu kojoj je cilj razjasniti područje programiranja aplikacija za .NET platformu i pružiti vam uvid u opsežnost tematike. Kako je Microsoftova platforma izuzetno popularna među programerima, trenutno su vam dostupne stotine knjiga koje obrađuju djelić tematike .NET-a – većina ih se bavi isključivo jednim aspektom programiranja u .NET-u, poput samog jezika C#, ASP.NET-a, ADO.NET-a ili pak izrade web-servisa. Da biste u potpunosti ovladali svime što vam .NET pruža, nema vam druge nego da pročitate cijeli niz knjiga.

Ova knjiga pak koristi drugačiji pristup: širokim rasponom tema dat će vam temelje zahvaljujući kojima ćete biti u stanju izrađivati kompletne aplikacije različitih tipova, od konzolnih, preko prozorskih i web-aplikacija, pa sve do web-servisa i aplikacija za mobilne uređaje. Naravno, zbog opsega tematike, sve teme nisu obrađene u potpunosti, već je naglasak stavljen na bitne i ključne dijelove, nužne za razumijevanje načina rada i korištenja pojedinih tehnologija. Bilo da iza sebe već imate poveće programersko iskustvo ili je ovo vaš prvi susret s programiranjem, u ovoj ćete knjizi naći sve potrebno da zaokružite svoje znanje u cjelinu koja će predstavljati idealnu podlogu na kojoj ćete moći dalje sami graditi i stjecati nova znanja o .NET-u.

Cijela knjiga temeljena je na velikom broju praktičnih primjera pisanih u C#-u. Naravno, i teorije ima napretek, no to ništa ne bi značilo bez kvalitetnih primjera uporabe neke tehnologije. Knjiga vas potiče i da sami isprobavate različite pristupe u programiranju .NET aplikacija stalno vam dajući ideje i koncepte koje možete sami dodatno razraditi korištenjem Visual Studija .NET čija probna verzija dolazi na DVD-u uz knjigu.

Što nećete naći u knjizi?

Knjiga neće ulaziti u detalje rada Interneta, korištenja računala i snalaženja u operacijskom sustavu na kojem programirate. U ovoj knjizi također nećete naći gomile dosadne teorije i hrpe primjera

koji pokazuju mogućnosti tehnologije, a nemaju neku praktičnu primjenu. Kako se o tematici ove knjige moglo napisati nekoliko svezaka knjiga, bilo je nužno izostaviti neke dijelove. Stoga se od vas očekuje da ćete nakon što pročitate pojedina poglavlja sami istraživati i, ukoliko vas zanima i imate potrebu za tim, otkriti dodatne mogućnosti.

Ideja je da vam svako poglavlje pruži najpotrebnije informacije pomoću kojih možete sami krenuti u programiranje. Postojala je opasnost da poglavlja koja se bave konkretnom tehnologijom počnu nalikovati priručnicima. To je izbjegnuto velikim brojem praktičnih primjera i primjene tehnologije, no u tom slučaju od njih ne možete očekivati da će poslužiti kao reference za same programske jezike ili dostupne objekte. Za takve stvari trebat ćete ipak konzultirati službenu dokumentaciju. To je i bit – u ovoj knjizi naučit ćete osnovne koncepte programiranja u .NET-u (iako ćete zahvaljujući tim "osnovama" biti u stanju izraditi složene i napredne aplikacije) te kako i naučiti čitati dokumentaciju i gdje potražiti pomoć. Zahvaljujući tome, nema šanse da se izgubite ili negdje zapnete, jer je pomoć nadohvat ruke.

Kako koristiti knjigu?

Prije nego što počnete čitati knjigu, važno je da se upoznate s načinom označavanja pojedinih dijelova teksta.

Način označavanja teksta

Samo srce ove knjige, izvorni kôd za sve primjere, označen je neproporcionalnim pismom i okomitom oznakom s lijeve strane. Naprimjer ovako:

```
for (i = 0; i < 10; i++) {
   MessageBox.Show("Dobar dan " + i.ToString() + ". put!");
}
```

Tipkovničke kratice spojene su znakom plus ("+"), što označava da se navedene tipke trebaju istovremeno pritisnuti da bi se obavila neka akcija ili uključila opcija, primjerice CTRL+ALT+N. Naredbe iz izbornika ispisane su kosim slovima, naprimjer *File - Open*. Na jednak su način ispisana i imena gumba iz dijaloških okvira i ikona, primjerice *Ok* ili *Cancel*. Kosim su slovima napisani i termini na stranim jezicima za koje ne postoji hrvatska riječ. Linkovi su pak podcrtani kako bi vam skrenuli pozornost.

Posebni okviri

Neke uže teme, koje katkada nisu neposredno povezane s okolnim tekstom, a ipak bi se mogle uvrstiti u dotično poglavlje te neke druge priče i anegdote izdvojene su u zasebne okvire. Prepoznat ćete ih po sivoj pozadini, kao na primjeru koji je na ovoj stranici.

.NET igrarije

Godine 1997. legendarna tvrtka za razvoj računalnih igara id Software objavila je novu verziju svoje tada najpopularnije igre, tzv. *first-person shootera*, Quake 2. Igra je postigla enorman uspjeh, prodana je u više od milijun kopija i zaradila je priznanje industrije kao igra godine. Krajem 2001. godine, id Software je darežljivo učinio izvorni kôd 3D sustava igre Quake 2 javno dostupnim svima zainteresiranima.

Sredinom 2003. godine to je odlučila iskoristiti tvrtka Vertigo Software i prevela je izvorni kôd pisan u jeziku C na .NET tehnologiju! Učinili su to samo zato da pokažu mogućnosti .NET-a, jer se sama igra sada vrti kao upravljana (*managed*) .NET aplikacija (kakve su to upravljane aplikacije saznat ćete čitajući samu knjigu) korištenjem Microsoft Common Language Runtimea, dijela .NET-a, bez ikakvih gubitka na performansama.

Naravno, to nam samo pokazuje mogućnosti .NET tehnologije. Da bi se i sami uvjerili u to, posjetite web-stranice na adresi http://www.vertigosoftware.com/Quake2.htm, gdje možete besplatno preuzeti kompletan izvorni kôd za igru Quake 2 pisan na .NET tehnologiji, učitati ga u Visual Studio .NET i pokrenuti samu igru. Ako želite, možete i mijenjati izvorni kôd i napraviti vlastitu verziju igre!

Savjeti, napomene i upozorenja posebno su istaknuti, "izvađeni" u zasebne okvire izvan teksta te označeni posebnim ikonama kako biste ih lakše uočili i kako biste shvatili koliko je važno da ih zapamtite. Upozorenja su, primijetit ćete, dodatno istaknuta jačim toniranjem pozadine na kojima su ispisana kako biste na njih obratili još veću pažnju nego na savjete i napomene.

Shvatili ste u čemu je štos sa savjetima, napomenama i upozorenjima, kao i popratnim okvirima, podebljanim slovima, kosim slovima i neproporcionalnim pismom. Tu su da vam olakšaju čitanje knjige, ali i da vam pomognu da se kasnije lakše snađete kad budete htjeli nešto pročitati ponovno.

Savjeti najčešće sadrže uputu koja će vam pomoći da neku radnju obavite brže, da upoznate neko izuzetno korisno svojstvo pri radu s raznim programima, da naučite nešto što bi vam moglo biti od posebne koristi ili jednostavno da vas spasi muka i lupanja glavom o zid pri rješavanju nekog problema. Naprimjer: "U prozoru Add Web Reference ne trebate ručno upisivati adresu web-servisa ukoliko je on smješten na istom računalu – jednostavno možete kliknuti na link *Browse to – Web services on the local machine* i dobit ćete popis svih web-servisa na računalu."

Napomene sadrže dodatne informacije koje bi vam mogle biti korisne. One više nadopunjavaju okolni tekst nego što predstavljaju njegov sastavni dio. Često se radi o važnim stvarima koje su ključne za ispravno korištenje pojedinih naredbi ili za razumijevanje koncepata. Primjerice: "Autentikacija je proces utvrđivanja *tko* je došao na stranice, a autorizacija je utvrđivanje ima li on *pravo* pristupa određenim stranicama."

Upozorenja vam skreću pažnju na određene opasnosti koje na vas vrebaju u nekim situacijama. Dok ste savjete i napomene mogli letimično pogledati, upozorenja nemojte preskakati. Naprimjer: "Klasa u .NET-u može nasljeđivati od samo jedne klase, koja se tad naziva *baznom* klasom. No primijetite da zato klasa može nasljeđivati od više sučelja (naravno, pritom ima obavezu implementirati sve metode svih naslijeđenih sučelja)."

Kontaktirajte autore

Ukoliko nas želite kontaktirati iz bilo kojeg razloga, pisati nam možete na sljedeće adrese:

domagoj.pavlesic@bug.hr
luka.abrus@bug.hr

Slobodno se obratite s dodatnim pitanjima, a posebno ćemo biti zahvalni na savjetima, kritikama i ukazivanju na eventualne pogreške ili propuste kako bismo ih mogli ispraviti u kasnijim izdanjima knjige.

Želimo vam uspješnu karijeru programera i mnogo kvalitetnih uradaka!

Domagoj Pavlešić i Luka Abrus

I. dio

.NET iznutra

1 POGLAVLJE

Uvod u .NET

U ovom poglavlju:

- Osnovni koncept i snaga .NET-a
- Budućnost .NET-a

Računala su nesumnjivo promijenila naš život, i njihovo poznavanje nužno je za obavljanje bilo kakvog posla. Internet i računala omogućavaju ono što smo dosad mogli samo sanjati – od korištenja bankarskih usluga i potpunog upravljanja svojim novcem do rezervacije kino-ulaznica i kupovine svakojakih proizvoda.

Dok je sav napredak na području računarstva iznesen na leđima računalnih zanesenjaka i tehnološki orijentiranih ljudi, popularizaciju svega ipak možemo zahvaliti isključivo svakodnevnoj upotrebljivosti dostupne tehnologije. Ona se sve više okreće *običnim* ljudima kojima računala nisu nužna za obavljanje svakodnevnog posla. I tu leži ključ popularizacije tehnologije: u trenutku kad više nije potrebno čekati satima u redu u banci, izlaziti s posla da bi se obišlo dućane ili dolaziti mnogo ranije u kino da bi se uhvatila dobra mjesta, svatko želi znati koristiti računala.

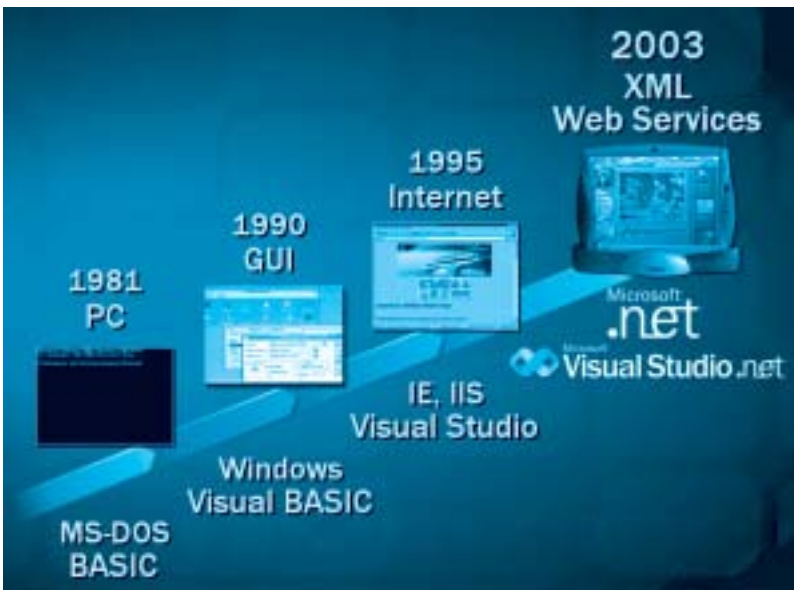
No opet, biti programer u takvom svijetu sve je teže. Aplikacije se nalaze na svakom koraku i međusobno se veoma razlikuju – od aplikacija za mobitele i aplikacija za bankomate, preko knjigovodstvenih aplikacija i web-dućana do standardnih prozorskih aplikacija za pisanje teksta i slanje *e-mailova*. Želite li se baviti razvojem aplikacija, morate biti upoznati s izuzetno velikim brojem tehnologija i tehnika programiranja, kojih je svakim danom sve više, a često već postojeća znanja nisu upotrebljiva u novim okruženjima.

To je pak, izuzetno važno – tvrtkama je vrlo teško u stopu pratiti nove tehnologije zbog ograničenog znanja. Skupo je stalno mijenjati fokus i proučavati različite tehnologije, pa mnogi idu linijom manjeg otpora i slijepo se drže onog što znaju i što im dobro ide. Tako nije rijedak slučaj da se i danas razvijaju tekstualne aplikacije u okruženju DOS-a za upravljanje dućanima i knjigovodstvom.

Pri razvoju novih tehnologija mnogo se polaže na iskoristivost dostupnog znanja. Ukoliko je programerima zahvaljujući trenutnom poznavanju programskih jezika i tehnologija vrlo lako usvojiti nove tehnike programiranja, svi su zadovoljni – i programer koji osjeća da napreduje – i tvrtka koja dobiva kvalitetnije ljude i može obavljati modernije projekte, i – na kraju – sam proizvođač tehnologije, zbog prihvaćenosti njegovih modela.

Koncept .NET-a

Počnimo s objašnjenjem .NET-a. Radi se o Microsoftovoj platformi za razvoj aplikacija koja predstavlja ujedinjenje različitih tehnologija i načina programiranja. Tijekom povijesti su se programeri orijentirani Microsoftu susretali s gomilom tehnologija, kao što su MS-DOS, Windows, OLE, COM, COM+, ActiveX, MFC, ASP…



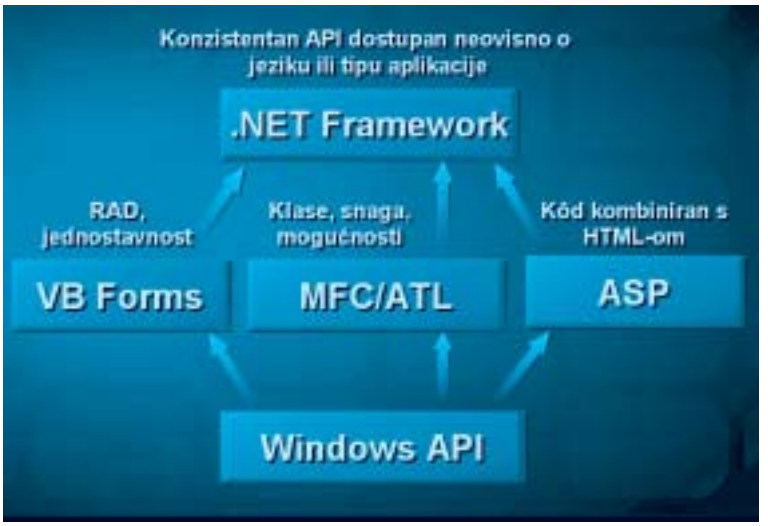
Slika 1-1: ***Razvoj programiranja tijekom godina***

.NET, dakle, predstavlja ujedinjenje različitih koncepata. Želite li raditi web-aplikacije? Nema problema, programirajte u .NET-u. Želite li raditi obične *desktop*-aplikacije? Nema problema, programirajte u .NET-u. Želite li raditi aplikacije za mobilne uređaje, kao što su dlanovnici ili SmartPhone uređaji? Nema problema, programirajte u .NET-u.

Za razvoj aplikacija za mobilne uređaje postoji posebna verzija .NET-a, nazvana .NET Compact Framework. Radi se verziji .NET-a iz koje su izbačene neke bazne klase koje nisu potrebne na mobilnim uređajima, no ona je inače u potpunosti funkcionalna i omogućava razvoj aplikacija za mobilne uređaje na jednak način kao i za Windowse.

Više nema potrebe učiti različite tehnologije – ako znate programirati web-aplikacije u .NET-u, tada znate raditi i prozorske aplikacije za Windowse. Tome je uvelike zaslužna unificirana razvojna okolina Visual Studio .NET koja sve to omogućava.

Visual Studio .NET predstavlja razvojnu okolinu za svaki tip .NET aplikacija, no i više od toga – mnogi Microsoftovi proizvodi koriste Visual Studio .NET kao razvojnu okolinu. Primjerice, želite li stvarati izvještaje koristeći Reporting Services dodatak za SQL Server 2000, koristit ćete Visual Studio .NET. Isto tako, želite li raditi s Commerce Serverom ili BizTalk Serverom, Visual Studio .NET postat će vam najbliži suradnik.



Slika 1-2: ***.NET predstavlja unifikaciju različitih programskih modela.***

Shvatili ste poantu, no to nije sve. .NET u sebi sadržava niz tehnologija, koje će redom biti objašnjene u ovoj knjizi, a sve one predstavljaju rezultat višegodišnjeg razvoja i unaprjeđenja postojećih tehnologija.

Višejezičnost

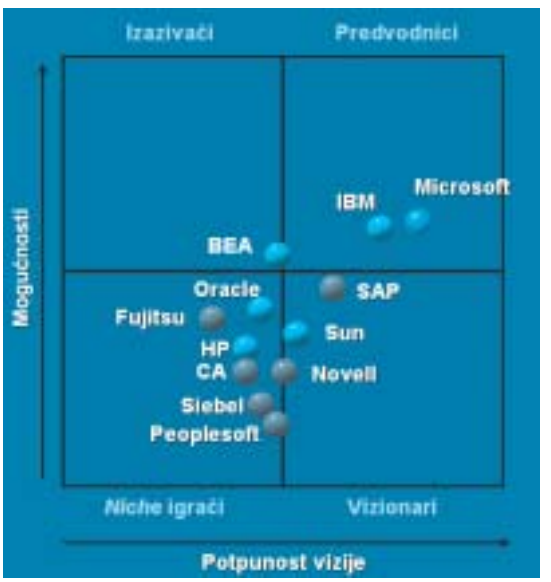
Vratimo li se na primjer s početka teksta, .NET je učinio ogroman korak u smjeru olakšavanja prvih programerskih koraka na novoj platformi. Microsoft je, naime, standardizirao osnovne dijelove .NET Frameworka (postali su dio ISO i ECMA standarda), što pak omogućava nezavisnim proizvođačima da nadograđuju mogućnosti, kao što su novi jezici kojima se može programirati u .NET-u.

Microsoft je napravio podršku za nekoliko jezika u .NET-u – VB.NET, C++.NET, J#, C# i JScript.NET – a zahvaljujući akademskoj zajednici i pojedinim industrijama, postoji više od 15 jezika u kojima se može programirati. Stoga, ukoliko ste iskusan programer u COBOL-u, Perlu, Fortranu, Eiffelu ili Smalltalku, bez straha – i vi možete lako početi programirati u .NET-u. Svima drugima, programerima s iskustvom na nekoj od Microsoftovih tehnologija, .NET će predstavljati samo korak naprijed.

Stoga za .NET kažemo da je višejezičan, jer omogućava programiranje u velikom broju jezika, a zahvaljujući javno dostupnim standardima i vi možete napraviti podršku za neki svoj jezik. Višejezičnost podrazumijeva istu razvojnu okolinu (Visual Studio .NET) za bilo koji od programskih jezika, isti *debugger* za pronalaženje problema i njihovo otklanjanje iz kôda aplikacija.

Web-servisi

.NET donosi novost u programiranju na području web-servisa. Odmah razjasnimo – radi se o prihvaćenom standardu za komunikaciju između aplikacija pomoću SOAP poruka prenošenih Internetom (više o svemu saznat ćete u kasnijim poglavljima knjige). No snagu web-servisa predstavlja mogućnost integracije različitih sustava i aplikacija na standardizirani način za komunikaciju.



Slika 1-3:
Slika prikazuje razmještaj platformi za razvoj web-servisa prema Gartnerovom izvješću iz 2003. godine. Važno je uočiti da se samo Microsoft i IBM nalaze u magičnom kvadrantu odnosno među vodećim proizvođačima (primijetite i da Microsoft vodi). Znakovita je i anketa CIO Magazina iz 2003, prema kojoj od 369 CIO osoba čak 46,5% preferira Microsoftovu platformu .NET za razvoj web-servisa (drugi je IBM-ov WebSphere s 19%).

I ostali proizvođači imaju razvojne alate i okruženja za razvoj web-servisa, no izrađivanje web-servisa uz pomoć .NET-a slovi za najjednostavniji i najmoćniji način. Iskoristivost web-servisa velika je i veoma raširena – koriste se za dohvaćanje različitih informacija, a mogu se koristiti u različitim aplikacijama (primjerice, Microsoft Office 2003 može koristiti web-servise iz svog *Research and Reference* prozora).

Slika 1-4: ***Web-servisi mogu uvelike olakšati razvoj aplikacija i njihovo povezivanje s drugim uslugama.***

Uporaba je jednostavna: u svojoj aplikaciji koju izrađujete za Visual Studio .NET-u jednostavno dodate referencu na neki web-servis, upišete njegovu adresu i možete ga pozivati iz svoje aplikacije kao da se radi o bilo kakvom drugom servisu koji se ne nalazi negdje na Internetu. Njihova izrada je također jednostavna – razvojno okruženje će najveći dio posla obaviti za vas, a vama ostaje samo izraditi funkcionalnost web-servisa odnosno metode koje će moći izvršavati. No strpite se još malo, sve vas to čeka kasnije u knjizi.

Snaga .NET-a

Danas se još uvelike koriste DLL-ovi i druge COM komponente koje sadržavaju neku funkcionalnost. S njima se radi drugačije i često predstavljaju problem – zahvaljujući .NET-u, programski modeli su se unificirali i sa svime se može komunicirati standardnom objektno orijentiranom funkcionalnošću.

Također, zahvaljujući CLR-u, središnjem dijelu .NET-a, više nema potrebe za mučenjem s dosad obaveznim elementima u Win32 i COM programiranju, kao što su GUID-ovi, HResult i sučelja IUnknown. Ti elementi nisu skriveni od programera, već jednostavno više ne postoje – CLR ih u potpunosti zamjenjuje.

Iako bi neupućeni promatrač na .NET mogao gledati kao na još jednu u nizu Microsoftovih tehnologija koje zahtijevaju isključivu posvećenost Microsoftovim proizvodima, uslugama i alatima, to nije točno. Naime, zbog već spomenute standardizacije CLR-a, .NET može biti implementiran na bilo kojoj platformi.

.NET je po nekim dijelovima sličan Javi – programi se prvo prevode u MSIL, a tek na ciljnom računalu pri prvom pokretanju se generira izvršni kôd. Iz toga slijedi logičan zaključak da se .NET aplikacije mogu izvršavati na bilo kojoj platformi na kojoj postoji verzija CLR-a kompatibilna sa standardom ECMA i bazne klase. Dakle, .NET može biti bez problema implementiran na IA64, Alpha, PowerPC računalima ili na bilo kojem operacijskom sustavu, kao što su MacOS, Linux ili Unix. Zapravo, u vrijeme pisanja ove knjige završava se projekt koji omogućava izvršavanje .NET aplikacija na Linuxu.

Kôd pisan u .NET-u često se naziva *managed* kôdom odnosno *upravljanim kôdom*, jer se izvršava u kontroliranom okruženju. Naravno, još uvijek možete pisati tzv. *unmanaged* kôd, no kako se .NET bude razvijao u narednim verzijama, to više neće biti moguće.

.NET se uvelike bazira i na XML-u, standardnom formatu za razmjenu podataka. Napredujući kroz ovu knjigu, uvidjet ćete da mnogi dijelovi .NET-a koriste baš XML za svoju internu uporabu, a ne treba ni spominjati web-servise, koji su u potpunosti temeljeni na XML-u.

S nabrajanjem snaga .NET-a mogli bismo nastaviti unedogled, stoga stanimo. Pravu snagu .NET-a uvidjet ćete čitajući naredna poglavlja ove knjige.

Budućnost .NET-a

Sve što ćete naučiti uz pomoć ove knjige i svih dostupnih resursa o .NET-u relativno je nevažno, ako .NET ne predstavlja važnu tehnologiju koju ćete moći još dugo vremena upotrebljavati. Naime, bilo bi potpuno besmisleno da utrošite godinu dana na svladavanje .NET-a i postanete vrhunski stručnjak pa doznate da Microsoft odustaje od daljnjeg korištenja .NET-a, da ga baca u sjenu i predstavlja neku novu tehnologiju.

Zahvaljujući svojoj snazi i poziciji na tržištu, Microsoft mora ponuditi plan razvitka svojih tehnologija i dugoročnu obavezu, a njegova je poruka jasna – .NET će bit s nama još jako dugo. Trenutno je aktualna verzija 1.1 .NET-a, a sljedeća (2.0) nosi kodno ime Whidbey. No Microsoft ima mnogo dugoročniji plan – sljedeća verzija klijentskog operacijskog sustava, kodnog imena Longhorn, čiji se izlazak planira u 2006. godini, u potpunosti će se bazirati na *managed* kodu odnosno na .NET-u.

Slika 1-5:
Razvoj koncepata programiranja vodi prema uslugama.

Zar uopće trebate bolju potvrdu Microsoftove predanosti .NET-u? Sama činjenica da će cijeli operacijski sustav biti izrađen oko .NET-a dovoljno govori o njegovoj uporabljivosti u stvarnom svijetu. Da biste se pripremili za vrijeme koje nam dolazi, pišite *managed* kôd i koristite .NET za stvaranje aplikacija.

Hrabro zakoračite u svijet .NET-a! Izabrali ste pravu programersku platformu, koja će vam omogućiti da lako razvijete sve svoje mogućnosti i stvarate aplikacije po vlastitoj želji. Web-aplikacije, web-servisi, *desktop* aplikacije, aplikacije za mobilne uređaje – izbor je samo vaš.

2. POGLAVLJE

Arhitektura .NET-a

U ovom poglavlju:

- Što je .NET Framework
- Od čega se sastoji .NET Framework
- Common Language Runtime
- Microsoft Intermediate Language
- Biblioteke klasa
- Metapodaci, *assemblyji* i manifesti

Nemojte se dati zastrašiti naslovom ovog poglavlja – riječ "arhitektura" u informatičkom smislu zvuči kompliciranije nego što zaista jest, a razumijevanje arhitekture .NET-a uvelike će vam olakšati napredovanje kroz knjigu i budući programerski život. Dobar dio poglavlja što slijede djelomično ili potpuno se naslanjaju na stvari koje će biti objašnjene na sljedećih nekoliko stranica, pa je uputno da ovo poglavlje "odradite" s maksimalnom pozornošću, koliko god vas vuklo da ga preskočite. Mi vam zauzvrat obećavamo da ćemo biti maksimalno jednostavni te da nećemo ići u nepotrebne detalje. Dogovoreno?

Iako se pod naslov "Arhitektura .NET-a" može ugurati zaista svašta, poglavlje će prvenstveno biti orijentirano na središnji dio .NET platforme poznat pod nazivom .NET Framework i seciranje njegove glavne komponente – Common Language Runtimea.

.NET Framework

Najjednostavnije rečeno, .NET Framework je dio koji nadograđuje mogućnosti samog OS-a. Radi se o posebnoj infrastrukturi koja programerima nudi gotova rješenja i funkcionalnosti da bi ubrzala i pojednostavila razvoj aplikacija svih vrsta i oblika. Naravno, kako programer za vrijeme razvoja koristi ono što mu .NET nudi, logično je da će i za izvršavanje te aplikacije biti potreban .NET Framework. Stoga treba zaključiti da je .NET Framework infrastruktura nužna i za razvoj i za izvršavanje aplikacija koje koriste .NET tehnologiju.

Da tehnologija bude što prije prihvaćena i korištena, bilo je potrebno omogućiti korištenje .NET Frameworka što većem broju korisnika. Kako je nerealno očekivati da će svi odmah prijeći na Windowse 2003 Server (s kojima, u trenutku pisanja jedinima, .NET Framework dolazi "u kutiji"), potrebno je bilo napraviti distributivnu inačicu .NET Frameworka koja se može instalirati i na starije operativne sustave.

Instalacija .NET Frameworka

Nema sumnje da će s vremenom svi operativni sustavi koji dolaze iz Microsoftovih radionica uključivati .NET Framework ili se čak i sami u potpunosti oslanjati na tu infrastrukturu. Sve inačice Windowsa izašle prije objave konačne verzije .NET Frameworka (što obuhvaća i Windowse XP) trebaju se ručno nadograditi. To možete učiniti pomoću instalacijske datoteke preuzete direktno s Microsoftovih web-stranica (http://www.microsoft.com/net) ili preko servisa Windows Update (http://windowsupdate.microsoft.com). Ipak, kako ste čitanjem ove knjige na putu prema programerskim vodama, vjerojatno ste instalaciju ovog dodatka već obavili ili ćete to napraviti u paketu s instalacijom Visual Studia .NET ili nekim drugim komadom softvera koji zahtijeva instaliran .NET Framework.

Slika 2-1: ***U inačicama Windowsa u kojima je potrebno doinstalirati .NET Framework, njegovo postojanje možete provjeriti u Control Panelu***

.NET Framework, kao i svaki drugi program, s vremenom se razvija i raste te dobiva nove brojčane oznake. U trenutku pisanja knjige aktualna inačica .NET Frameworka je 1.1 i nužna je za rad s Visual Studiom .NET 2003. Oko kompatibilnosti ne morate brinuti – na istom računalu može koegzistirati više različitih inačica.

U teoriji, .NET Framework bi se mogao staviti na računalo s bilo kojim operativnim sustavom. Podrška za Windowsove operativne sustave (od Windowsa 98 nadalje) razvijena je od samog Microsofta, dok se to za ostale platforme očekuje od drugih kompanija ili organizacija.

Prva se tog posla primila kompanija Ximian (http://www.ximian.com) pokrenuvši projekt nazvan Mono. Cilj projekta je izraditi *open-source* implementaciju .NET Frameworka koja bi radila na Linuxu i drugim *unixoidnim* operativnim sustavima. Kako im ide i što su sve dosada napravili možete pratiti na stranicama http://www.go-mono.com. Projekt je vrijedan pohvale, no ostaje da se vidi hoće li Microsoftova inicijativa zaista zaživjeti na drugim platformama.

Od čega se sastoji .NET Framework?

Kako je .NET Framework osnova .NET-a, sasvim je logično da će veći dio knjige govoriti o stvarima koje su dio .NET Frameworka. Ipak, prije nego što krenemo na detaljnu razradu, treba sagledati kompletnu sliku i vidjeti na koji način su dijelovi međusobno povezani.

Najvažnija sastavnica .NET Frameworka zove se Common Language Runtime (naravno, nikada je u javnosti nemojte tako nazivati – uvijek koristite skraćeni oblik CLR, čitan prema engleskim pravilima sricanja). Usporedimo li .NET s računalom, CLR bi zauzeo ulogu procesora – čipa koji upravlja svime što se u računalu zbiva.

Dakako, CLR nije čip već softverski sustav u kojem se kôd pokreće i izvršava. Kada pokrenete program koji je pisan za .NET platformu, CLR ga učitava i pokreće u sebi kako bi mu osigurao stabilnost i sigurnost. Instrukcije u programu se u realnom vremenu (dakle u trenutku kada je program pokrenut) prevode u izvorni mašinski kôd (*engl. native machine code*) koji razumije računalo. Za taj je posao zadužen JIT-kompajler, što je skraćenica za *just in time*. Kako se u većini slučajeva radi o x86 kompatibilnim računalima, tako je i kôd koji JIT-kompajler generira onaj koji razumiju takva računala. Upravo zbog prevođenja u izvorni mašinski kôd računala postoji mogućnost da .NET Framework bude prenesen i na druge operativne sustave koji ne dolaze iz Microsoftovih radionica.

Dobro ste zaključili da takav način rada osjetno usporava izvršavanje aplikacije. Kompiliranje kôda, ma koliko brzo bilo, ipak zahtijeva određeno vrijeme, što će krajnji korisnik doživjeti kao sporo učitavanje aplikacije. Stoga se kompiliranje vrši samo jednom, a njegov rezultat sprema kako bi

se kasnije mogao koristiti bez ponovnog prolaska kroz proces kompiliranja. Spremljeni kôd ponovo se kompilira tek ako se nešto u aplikaciji promijeni.

Kôd koji se izvodi unutar CLR-a nazivamo upravljani kôd (engl. *managed code*). Oni napredniji, kada ih na to natjeraju prilike, mogu pisati i/ili koristiti neupravljani kôd (engl. *unmanaged code*) koji neće biti izvršavan unutar CLR okruženja. Neupravljanim kôdom, dakle, zovemo i sve "stare" module, poput COM-komponenti.

Kao što je poznato, aplikacije za .NET platformu možete pisati u raznim programskim jezicima, gotovo svim poznatijim. CLR, međutim, ne poznaje niti jedan taj jezik – njemu naredbe dolaze isključivo u jeziku koji se zove Microsoft Intermediate Language (skraćeno MSIL, IL, a u starijoj literaturi CIL) koji je temeljen na pravilima koja se nazivaju Common Language Specification (CLS). Zaključak je jasan – mora postojati kompajler koji će programski jezik u kojem čovjek programira prevesti u MSIL kako bi ga razumio CLR.

Takvi kompajleri nazivaju se IL-kompajleri i postoje za velik broj jezika. Microsoft je napisao kompajlere za pet jezika: C#, J#, C++, Visual Basic i JScript, a ostali proizvođači softvera potrudili

Kako izgleda taj MSIL?

Za one koji jednostavno moraju vidjeti kako izgleda MSIL, evo kratkog primjera:

```
ldc.i4.4
stloc.0
ldc.i4.5
stloc.1
ldloc.0
ldloc.1
add
stloc.2
```

U prvom redu uzimamo brojku četiri i u drugom je spremamo u varijablu broj 0. U sljedeća dva reda na isti način varijabli broj 1 pridružujemo vrijednost 5. Zatim uzimamo te dvije varijable instrukcijama *ldloc* i zbrajamo ih. Dobiveni zbroj spremamo u varijablu označenu brojem 2.

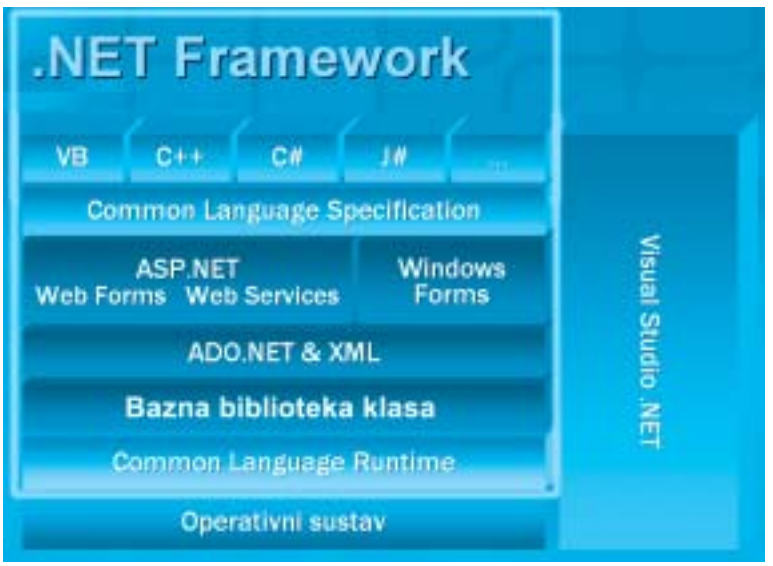
Ukoliko imate dovoljno vremena i volje za ovakvo programiranje, ili jednostavno želite ostaviti IL-kompajlere bez posla, samo izvolite. Mi ćemo se radije držati klasičnih, čovjeku razumljivijih jezika kao što je *C#*, u kojem gornji primjer izgleda višestruko jednostavnije:

```
int a = 4;
int b = 5;
int c = a + b;
```

su se oko brojnih drugih, uključujući Perl, Python, Cobol i Eiffel. Teoretski, koji god jezik koristili moći ćete napisati jednako dobre i brze aplikacije. Ipak, izaberete li neki "egzotičan" jezik, u praksi ćete imati prilično problema oko razumijevanja primjera, kako u dokumentaciji tako i na brojnim web-stranicama i grupama na Usenetu.

Iako vam se na prvi pogled MSIL može činiti kao nepotreban korak, on je direktno zaslužan za velik dio komfora koji pruža programiranje u .NET okruženju. Zahvaljujući činjenici da se sav kôd prevodi u unificirani oblik, moguće su stvari o kojima dosada nismo niti sanjali, poput nasljeđivanja klasa bez obzira na to u kojem jeziku su napisane ili *debugiranje* kompletnog rješenja bez obzira na to što su njegove komponente pisane u različitim jezicima (o čemu će biti više govora u drugom dijelu knjige).

Mogućnosti koje CLR nudi izuzetne su, no same po sebi nisu dovoljno jednostavne i upotrebljive iz ljudskog aspekta. Pisanje programa u takvom okruženju bilo bi srednjovjekovno mučenje (s time da u informatici srednjim vijekom smatramo sedamdesete godine dvadesetog stoljeća). Stoga u .NET Frameworku postoje setovi klasa, gotovih funkcionalnosti, koje omogućavaju jednostavno i brzo iskorištavanje mogućnosti koje nudi CLR i mnogih često upotrebljavanih radnji.



Slika 2-2: ***Shematski prikaz dijelova .NET Frameworka***

Prva skupina klasa zove se bazna biblioteka klasa (engl. *Base Class Libary*, skraćeno BCL) i sadrži osnovne funkcionalnosti koje koristimo u programiranju. Primjerice, trebate li u svoju aplikaciju uključiti funkcije za transformaciju teksta, mrežnu komunikaciju, provjeravanje sigurnosnih prava ili hvatanje unosa s tipkovnice, koristit ćete funkcionalnosti koje se nalaze u ovoj biblioteci.

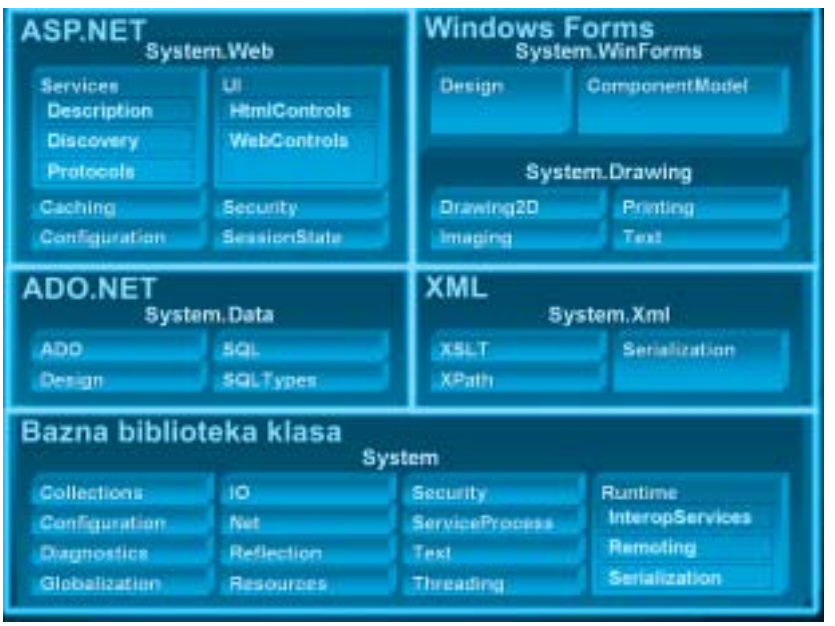
Svojevrsna nadogradnja osnovne biblioteke je ona koja sadrži set klasa zaduženih za suradnju s bazama podataka (ADO.NET) i XML-om. One nam omogućavaju povezivanje aplikacija s bazama podataka (poput SQL Servera), kao i manipulaciju podacima u XML dokumentima.

Dodatnih klasa ima još. Ukoliko se odlučite za razvoj klasične Windows aplikacije, koristit ćete klase okupljene pod imenom Windows Forms koje pružaju funkcionalnosti poput kreiranja prozora, izbornika, natpisa, polja za unos i sličnih objekata koje viđamo u klasičnim Windows aplikacijama. Naravno, klase se ne zaustavljaju samo na vizualnim detaljima, već omogućavaju upravljanje i kontrolu brojnih drugih "nevidljivih" sistemskih parametara.

S druge strane, skup klasa nazvan ASP.NET omogućit će vam razvoj dva tipa aplikacija – Web Forms i Web Services. Web Forms predstavljaju nadogradnju ili, bolje rečeno, zamjenu za serversko skriptiranje na web-stranicama (za što se prije .NET-a u Microsoftovim svijetu koristio ASP). Kažemo "zamjenu" jer Web Forms puno više sliči razvoju prozorskih aplikacija nego nadogradnji serverskog skriptiranja kakav smo koristili prije dolaska .NET-a.

Web-servisi predstavljaju web-aplikacije koje pružaju određenu funkcionalnost udaljenim programima. Primjerice, web-servis jedne banke može imati funkcionalnost preračunavanja iznosa iz eura u kune – vi njemu pošaljete iznos od 100 eura, a on vama vrati iznos od 770 kuna. Dakako, ovo je tek najjednostavniji primjer...

Radi jednostavnijeg snalaženja među klasama, one su raspoređene u hijerarhijsku strukturu koju možete vidjeti na slici uz tekst. Svaka stavka u strukturi ima svoje ime, koje nazivamo *name-*



Slika 2-3: ***Shema biblioteka klasa u .NET-u zajedno s pripadajućim namespaceovima***

space. Kao što u strukturi mapa na disku moramo znati točnu putanju da bismo došli do neke datoteke, tako i moramo znati točan *namespace* da bismo došli do određene klase. Evo primjera za dvije klase koje se jednako zovu, no nalaze se na drugim mjestima u hijerarhiji i, sukladno tome, imaju drugačije funkcije:

```
Using System.Web.UI.Control
Using System.Windows.Forms.Control
```

Da bi nama programerima sve te mogućnosti bile nadohvat ruke i jednostavne za korištenje, postoji alat nazvan Visual Studio .NET. On sam po sebi ne pripada .NET Frameworku, no u potpunosti se na njega oslanja. U njegovu izvrsnom sučelju provodit ćete najviše vremena, osim ako za svoj razvojni alat ne izaberete neki drugi koji ima mogućnost rada u .NET okruženju, kao što je to Borlandov C# Builder.

Organizacija kôda

Uzmimo, primjera radi, da smo već napredni programeri i da smo napisali svoj prvi program. Nakon kompiliranja, rezultat koji smo dobili jest izvršna datoteka koja najčešće ima nastavak *.exe*, *.dll* ili *.netmodule*. Takva se izvršna datoteka zove upravljani modul (engl. *managed module*) i sastoji se od četiri osnovna elementa.

Da bi se spomenuta datoteka mogla izvršavati unutar Windows okruženja, njen prvi dio mora biti u formatu koji se zove Portable Executable (skraćeno PE). Osnovna uloga tog dijela je reći operativnom sustavu da se radi o upravljanom modulu te prebaciti kontrolu izvršavanja na CLR. Slijedi dio koji se naziva CLR zaglavlje, u kojem se zapisuju osnovni podaci o modulu. Treći dio čini MSIL-kôd, dok četvrti sadrži podatke o podacima, tzv. metapodatke.

Metapodaci

Da bi se određeni modul mogao koristiti, moramo znati što se u njemu nalazi i kako je sadržaj unutar njega organiziran. Tu u priču ulaze metapodaci koji opisuju modul i njegov sadržaj te na taj način uvelike olakšavaju korištenje modula.

U vremenu prije .NET-a nije postojala tako strogo definirana potreba za podacima koji opisuju podatke. Iako je u određenim slučajevima postojala ta mogućnost, ona nikad nije bila toliko opširna i, što je još važnije, nužna. Naime, metapodaci nisu opcionalna stvar – svaki upravljani modul mora sadržavati metapodatke, i na taj način svima zainteresiranima pružiti detaljnu informaciju o sebi.

I za interoperabilnost među programskim jezicima, koju smo već spominjali u kontekstu MSIL-a, vrlo je važno postojanje metapodataka. Oni, zahvaljujući standardiziranom načinu opisivanja modula,

omogućavaju da moduli pisani u jednom jeziku koriste funkcionalnosti drugog, pisanog u tom ili nekom drugom jeziku.

Metapodaci se, osim za komunikaciju između modula, koriste i u gotovo svakom kutku .NET infrastrukture. Dio sistema koji je vjerojatno najviše zainteresiran za metapodatke je sam CLR. On pomoću njih radi provjeru valjanosti modula, aplicira sigurnosne parametre, planira korištenja memorije, povezuje s ostalim klasama koje modul koristi i na kraju pokreće izvršavanje. Bez metapodataka cijeli bi proces učitavanja i izvršavanja modula bio u najmanju ruku značajno složeniji i sporiji, ako ne i nemoguć.

Direktne koristi od metapodataka imaju i programeri. Primjerice, alati poput Visual Studia .NET koriste ih kako bi omogućili funkcionalnost automatskog kompletiranja riječi prilikom tipkanja koda, čime se smanjuje potreba za korištenjem dokumentacijom i pamćenjem brojnih naredbi.

Assemblies

Ukoliko naš upravljani modul ima određenu funkcionalnost koja će sama imati neku logičnu funkciju, onda ga istovremeno možemo zvati i *assembly.* Većina se *assemblyja* sastoji od samo jednog upravljanog modula, no postoje i oni koji u sebi ne samo da sadrže više upravljanih modula već i druge datoteke koje nisu moduli.

Najčešći slučaj u kojem nastaju *assemblyji* s više datoteka jest kada dva ili više modula pisanih u različitim jezicima povezujemo u jednu funkcionalnu cjelinu. Ta funkcionalna cjelina često ima potrebu i za nekim drugim resursima, poput slikovnih datoteka, koji također postaju dio *assemblyja.*

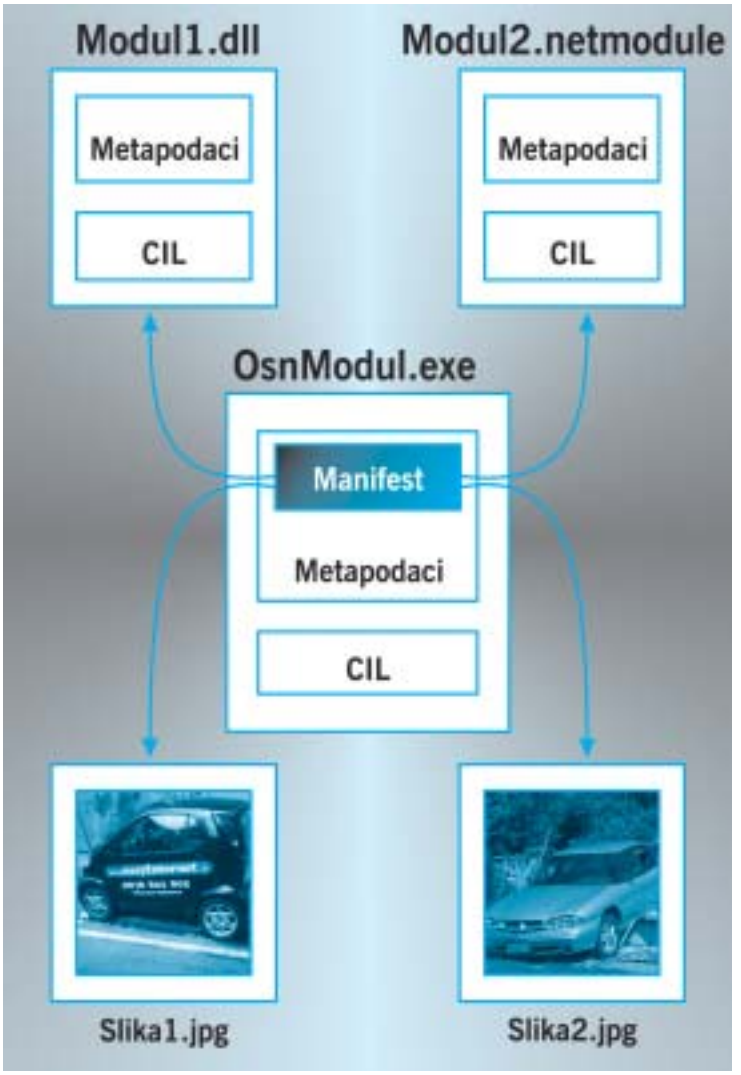
Prije nego što se izgubite u apstrakciji pojma "funkcionalna cjelina", objasnimo što konkretno obuhvaća pojam *assemblyja*. Najlakše objašnjiva kategorija jest aplikacija. Dakle, u .NET terminologiji, aplikacija poput Notepada ili Microsoft Worda (pod uvjetom da je pisana u .NET-u) smatrala bi se *assemblyjem*.

Nešto što smo dosada nazivali komponentama također pripada pod pojam *assemblyja*. Za razliku od prije spomenutih aplikacija, komponente najčešće nemaju korisničko sučelje i krajnji ih korisnici ne primjećuju, no zato aktivno sudjeluju u radu aplikacije koja ih koristi.

Primjer takvog odnosa može biti komponenta za provjeru pravopisa u aplikaciji za pisanje teksta. Iako se krajnjem korisniku to ne čini, radi se o dvije funkcionalne cjeline koje su međusobno odvojene i samostalne. Sve funkcionalnosti aplikacije (osim, dakako, provjere pravopisa) radit će i ako komponenta ne postoji, a funkcionalnosti komponente moći će koristiti i neka druga aplikacija, neka druga komponenta ili, sažeto sročeno, neki drugi *assembly.*

Manifesti

Da bi se znalo koje datoteke pripadaju u neki *assembly,* jedna od njih mora imati podatke o njemu i njegovim dijelovima. Ti se podaci nazivaju manifest i zapravo predstavljaju metapodatke o *assemblyjima.*



Slika 2-4: ***Shema assemblyja sastavljenog od tri upravljana modula i dvije slikovne datoteke***

Među tim podacima možemo pronaći ime *assemblyja* i listu datoteka koje čine *assembly*. Sadržaj svake datoteke je transformiran u šifrirani kontrolni izraz pomoću kojeg se može otkriti je li sadržaj datoteke mijenjan. Manifest sadrži i kopiju dijela metapodataka iz modula koje sadrži zajedno s informacijom na koji se modul odnose. Na taj način preko manifesta možemo direktno pristupiti funkcionalnostima pojedinih modula, i ne znajući da postoje.

Jedan od važnijih podataka koji se sprema u manifest jest njegova verzija. Ona se označava u formatu *major_version.minor_version.build.revision,* što u konačnici izgleda otprilike ovako: 1.2.2012.0. Osim broja verzije, tu može biti i takozvani *culture string* koji predstavlja zemlju i/ili jezik kojoj

je *assembly* namijenjen (kulturni string za Hrvatsku je "hr-HR"), a osim njega možemo još pronaći podatke poput imena autora, opisa, potrebnih sigurnosnih prava i slično.

Želite li pregledati koje metapodatke i/ili manifeste sadrži neka datoteka, poslužite se pomoćnim programima koji dolaze uz .NET Framework SDK. Više o tim alatima potražite u dodacima na kraju knjige...

Postojanje manifesta u *assemblyju* donosi još jednu izvanrednu "nuspojavu". Zahvaljujući činjenici da se u njemu nalaze svi podaci o *assemblyju*, nema potrebe za njihovom registracijom u sustav, kao što je to nužno s COM komponentama. Dovoljno ih je smjestiti na pravo mjesto na disku i mogu se koristiti.

Slabo i jako imenovanje

Imenovanje *assemblyja* posebna je priča. Imenovanje može biti slabo (engl. *weakly named*) i jako (engl. *strongly named)*. Slabo imenovani *assembly* je onaj koji nije digitalno potpisan, a na njega se referencira koristeći samo ime navedeno u manifestu, što nije ništa drugo nego ime datoteke u kojoj se manifest nalazi napisano bez ekstenzije.

Jako imenovani *assemblyji* digitalno su potpisani, a na njih se referencira pomoću kombinacije imena, javnog ključa, oznake verzije i *culture stringa* ukoliko isti postoji. Bilo kakva promjena u tim parametrima motivirat će CLR da tretira *assembly* kao sasvim nov, ma koliko male stvarne promjene bile.

Uzmimo za primjer aplikaciju koja koristi slabi *assembly* u kojem ste primijetili neki propust. Otvarate ga u razvojnom alatu, popravljate uočenu pogrešku, kompilirate ga i dobivenu izvršnu datoteku snimate preko stare, i stvar radi dalje bez imalo problema.

Ukoliko se u istoj situaciji nađe jaki *assembly,* njegova će zamjena biti nešto kompliciranija. Naime, ako stavite novu inačicu, aplikacija će i dalje koristiti staru, a u slučaju da potonju izbrišete, aplikacija će otkazati poslušnost jer neće moći pronaći verziju *assemblyja* koju koristi.

S druge strane, situacija starijim programerima poznata pod nazivom "DLL Hell", zahvaljujući jakom imenovanju, više ne postoji. Događa se, naime, da instalacija nekog novog softverskog paketa zamijeni neku od dijeljenih DLL-komponenata novom inačicom iste. U dobrom dijelu slučajeva ta zamjena prođe bez posljedica, no zna se dogoditi da neka prije instalirana aplikacija koja koristi tu komponentu otkaže poslušnost. To se događa u slučaju da nova verzija komponente nema sve mogućnosti koje su postojale u staroj ili su one tako nadograđene da ih se ne može koristiti

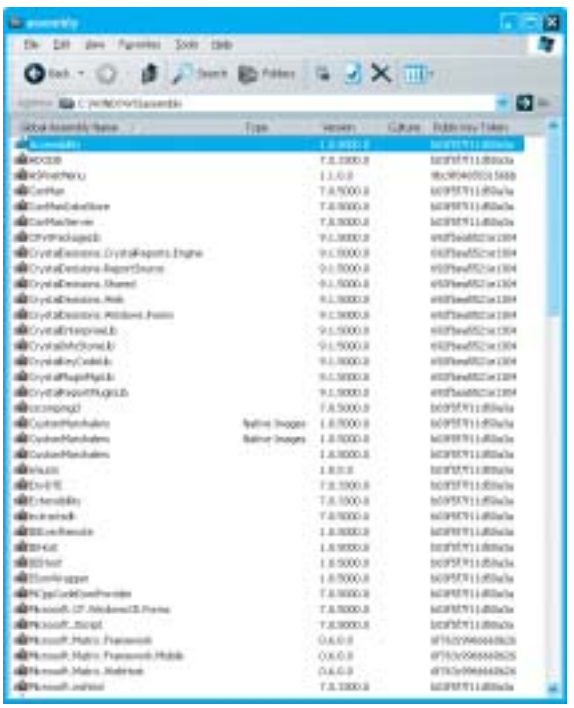
na isti način. Kako više nema funkcionalnosti koje stara aplikacija koristi, ona javlja grešku i otkazuje poslušnost. Pokušate li novosnimljenu inačicu komponente zamijeniti starom, nova aplikacija neće raditi. Bezizlazna situacija u kojoj se tada nalazite zove se pakao DLL-a.

U .NET okruženju nema takvih problema. Zahvaljujući jakom imenovanju *assemblyja*, sistem dozvoljava koegzistenciju više različitih verzija iste komponente – svaka će aplikacija nastaviti koristiti svoju verziju, a vi nećete vidjeti niti plamičak vječne vatre...

Kao što vidite, oba tipa imenovanja *assemblyja* imaju svoje dobre i loše strane. Kako je na programeru da izabere koje će imenovanje koristiti, stvar ćete moći prilagoditi okolnostima. S druge strane, vrlo je vjerojatno da ćete u početku razvijati isključivo slabe *assemblyje*, a kasnije, kada se dohvatite jakih, verzioniranje vam više neće biti problem.

Global Assembly Cache

Tip imena za svojeg ljubimca ne možete birati kada ga želite podijeliti s ostatkom svijeta. Drugim riječima, želite li napraviti *assembly* koji će moći koristiti sve .NET aplikacije na računalu, morat ćete ga digitalno potpisati i staviti na mjesto koje se zove globalni spremnik *assemblyja* (engl. *Global Assembly Cache*, skraćeno GAC). GAC je svojevrsno spremište svih javno dostupnih *assemblyja* (pod "javno" se misli na lokalno računalo).



Slika 2-5:
Pogled na Global Assembly Cache i neke assemblyje koji se u njemu nalaze

Razlog zbog kojeg GAC ne tolerira slabo imenovane *assemblyje* jest realna opasnost da se na istom računalu susretnu dva *assemblyja* istog (slabog) imena, a različitih proizvođača i funkcionalnosti.

Sada kada razumijemo pojam *assemblyja*, možemo reći da su sve one klase spomenute u priči o Base Class Libraryju i ostalim klasama u .NET Frameworku raspoređene po *assemblyjima* i da se nalaze u GAC-u. Zahvaljujući činjenici da su svi ti "ugrađeni" *assemblyji* jako imenovani, na istom se računalu može istovremeno instalirati i koristiti više inačica .NET Frameworka.

Garbage collection

Iako se radi o prilično naprednom sustavu koji po logici stvari ne bi trebao dobiti prostor u ovom letimičnom pregledu arhitekture .NET-a, ipak ćemo ga spomenuti. On, naime, može poslužiti kao izvrstan primjer prednosti upravljanog izvršavanja kôda unutar okruženja kao što je CLR.

Jedna od najnapornijih i dosadnijih zadaća programera je traženje tzv. *memory leaks*, mjesta gdje određeni objekti zauzimaju memoriju iako se više ne koriste i bez posljedica bi mogli iz nje biti uklonjeni.

Takvo traženje igle u plastu sijena u .NET okruženju postaje stvar prošlosti. Za to se brine sustav zvan Garbage collection koji pregledava memoriju kad se napuni i iz nje briše stvari koje se više ne koriste. Automatski, i bez zahtjeva programera, dakako.

Proces počinje time što se sav sadržaj memorije proglašava nepotrebnim. Zatim kreće šetnja kroz aplikaciju i stvaranje liste svih dijelova memorije koji se koriste. Kad je lista gotova, preostali, nepopisani dio memorije smatra se nepotrebnim i biva obrisan. Djelići memorije koji su preživjeli čišćenje skupljaju se u hrpu na početku memorije, vrlo slično načinu na koji Windowsi rade defragmentaciju tvrdog diska.

Kao što smo već spomenuli, ovo je tek jedna od beneficija izvršavanja koda unutar upravljanog okruženja. CLR se, osim skupljanja smeća po memoriji, brine još za pregršt drugih manjih ili većih stvari, sve s ciljem da programiranje učini jednostavnijim i bržim.

3. POGLAVLJE

Programski jezici

U ovom poglavlju:

- Osnovno o programskim jezicima
- Jezici dostupni u .NET-u
- Vaš prvi program
- Vodič kroz sintaksu jezika VB.NET i C#

Opće je poznato da svaki programer ima preferirani programski jezik koji koristi, voli i u koji se kune. Iako svaki od jezika ima svojih pozitivnih i negativnih strana, omiljeni programski jezik bira na temelju slučaja i okolnosti u kojima počinjete programirati. Mi vam tu ne dajemo neki izbor – cijela knjiga, izuzevši ovo poglavlje, bavit će se isključivo C# programskim jezikom, što iz uredničko-tehničkih razloga (nisu nam dali pisati knjigu od tisuću stranica), što iz političko-religijskih (i mi imamo svoj omiljeni jezik). Naravno, postoji i objektivni razlog za favoriziranje – radi se o jeziku koji je stvoren upravo za .NET platformu, pokušavajući sintetizirati jednostavnost Visual Basica i snagu C++-a.

Ipak, u ovom poglavlju ćemo se detaljno pozabaviti i sintaksom jezika Visual Basic .NET kako bismo onima kojima su bliže osnove toga jezika omogućili bezbolniji prelazak na novu, "ceoliku" sintaksu. Ne pokušavamo vas obratiti na novu "religiju" – samo želimo napomenuti da je novi VB.NET značajno promijenjen u odnosu na stari VB te da je učenje tih novotarija idealna prilika za prihvaćanje još poneke sitnice i prelazak na C#.

Imajte na umu da se .NET Framework prema svim programskim jezicima odnosi ravnopravno te nema tehničkih razloga zbog kojih ne biste programirali u nekom kronično nepopularnom jeziku kao što je, recimo, Haskell. Stoga ćemo prije seciranja sintakse dvaju favoriziranih spomenuti riječ-dvije o najpoznatijim "ostalim" jezicima podržanima u .NET-u.

Jezici u .NET-u

Visual Studio .NET dolazi s podrškom za četiri programska jezika. To su Visual Basic .NET, Visual C++ .NET, Visual C# .NET i Visual J# .NET. Izbor jezika strogo je "političke" naravi – svaki od nabrojanih jezika ima svoju zadaću u Microsoftovu marketinškom planu: VB kao mamac za početnike, C++ kao udica za iskusne programere, J# za brojne ljubitelje programiranja u Javi i C# – za sve ostale.

Prilikom instalacije Visual Studia možete odabrati koji od tih jezika želite koristiti. Naravno, izbor nije ograničen isključivo na nabrojane jezike. Postoji niz drugih jezika za koju su podršku razvili ostali proizvođači softvera, i koje možete instalirati kao dodatak Visual Studiju te koristiti u programiranju. Neki od tih dodataka imaju svoju cijenu, dok su neki besplatni, no u svakom slučaju o alternativnim jezicima razmišljajte tek ukoliko želite iskoristiti već postojeći kôd – za svako programiranje "od nule" preporučujemo prihvaćanje jednog od "originalnih" jezika.

Razlog tomu je jednostavan – zapnete li negdje, što će se sigurno dogoditi, potražit ćete pomoć od prijatelja ili na Internetu. Kako većina ljudi programira u nekoliko najpoznatijih jezika, primjer koji rješava problem najlakše ćete pronaći u njima, dok ćete za ostale morati dobiveno rješenje "prevoditi", ako uopće tako nešto bude moguće.

Osim tisuća primjera na grupama Useneta i raznim web-stranicama, najbolji argument za netom izrečenu konstataciju je originalna Microsoftova dokumentacija koja u većini slučajeva primjere navodi u dva jezika – VB.NET i C#.

Karakteristike jezika u .NET-u

Osnovna i najvažnija karakteristika svakog jezika koji želi raditi u okruženju .NET-a jest objektna orijentiranost. Više o tom pojmu i općenito objektno orijentiranom programiranju čitat ćete u šestom poglavlju, pa ovdje nećemo previše duljiti.

Kao što smo već natuknuli u prošlom poglavlju, bez obzira na jezik koji koristite, služit ćete se bibliotekama klasa ugrađenim u sâm .NET Framework. Korištenje istih biblioteka jezike međusobno čini puno sličnijima, a paralelno programiranje u više njih jednostavnijim. Štoviše, ta će vam karakteristika omogućiti da "pročitate" kôd pisan u jeziku čiju sintaksu možda niti ne poznajete.

Odgovor na vječno pitanje – koji jezik izabrati – u okruženju .NET-a nešto je jednostavnije dati. Stvari koje više ne mogu utjecati na odluku su brzina izvršavanja i proširivost. Bez obzira u kojem jeziku

U početku bijaše naredba...

Prvo "nešto" što bi se moglo nazvati pretečom programskih jezika koje danas poznajemo i volimo nastalo je davne 1946. u bavarskim Alpama, skrovištu njemačkog inženjera Konrada Zusea. Svoj je jezik nazvao Plankalkul, no on nije bio korišten u elektroničkim računalnim uređajima. Prvi takav stvoren je 1949. i nazvan je Short Code.

U godinama nakon toga rodilo se nekoliko novih programskih jezika, a jedan od njih, stvoren 1957. godine, bio je Fortran, najstariji i danas živući jezik, ujedno preteča većine današnjih jezika.

Riječ "Basic" zapravo je akronim izraza *Beginner's All-purpose Symbolic Instruction Code*. Napravljen je 1963. godine na koledžu Dartmouth, a tvorci su mu John G. Kemeny i Thomas E. Kurtz. Godinu dana kasnije je prvi put korišten na računalu IBM 704. Zahvaljujući jednostavnosti, jezik je vrlo brzo stekao veliku popularnost i implementiran je na brojne operativne sustave i strojeve, uključujući i svojevremeno izuzetno popularne Commodore 64 i ZX Spectrum.

Basic je od početaka bio vrlo zanimljiv Microsoftu, te ga je koristio na nekoliko načina i oblika. Uz DOS je neko vrijeme dolazio QBasic, a Windowse je u stopu pratio pod imenom Visual Basic, u nekoliko inačica. Postoji i skriptna inačica jezika nazvana VBScript koja je podržana u Windows Scripting Hostu (WSH) za skripte u Windowsima, ASP-u za serversko te Internet Exploreru za klijentsko skriptiranje web-stranica.

Tvorac jezika C bio je Dennis Ritchie iz Bell Labsa, daleke 1972. godine. Nećete nam vjerovati, no postojao je i jezik B koji je bio preteča C-a. Nemojte pogađati – njihov se prethodnik nije zvao A, već BCPL, a obiteljsko stablo seže sve do Algola i već spomenutog Fortrana.

Svoju popularnost jezik C duguje ponajprije Unixu zahvaljujući kome se nastanio u srca brojnih programera. Ipak, danas je broj ljudi koji programira baš u C-u manji – većina ih se koristi objektno orijentiranom inačicom C-a nazvanom C++. (Čisto kao zanimljivost – postoji i programski jezik D, nasljednik dvaju spomenutih C-a, koji nije previše zainteresirao programere, no još ima šanse – razvijen je 2003.)

Na temeljima C-a razvijen je, među ostalim, i objektan programski jezik Java. Njega je razvio James Gosling iz kompanije Sun Microsystems, a osnovni adut – portabilnost među platformama – donio mu je veliku popularnost među programerima, posebno onima nevezanim uz Microsoft.

JavaScript, skriptni jezik razvijen od strane tvrtke Netscape, služi za klijentsko skriptiranje u web-preglednicima i razvijan je nevezano s Javom iako s njom dijeli slično ime i sintaksu. Postoji i Microsoftova varijacija na temu JavaScripta nazvana JScript.

Postoji nekoliko tisuća programskih jezika, što mrtvih što živih, a vjerojatno najduži (iako ne i najažurniji) popis možete potražiti na adresi:

`http://people.ku.edu/~nkinners/LangList/Extras/langlist.htm`

Tablica 3-1:
Popis trenutno dostupnih alternativnih jezika u .NET-u koje su razvili nezavisni proizvođači

Jezik	Naziv	Adresa
Ada	Ada	http://www.usafa.af.mil/dfcs/bios/mcc_html/a_sharp.html
APL	Dyalog APL	http://www.dyadic.com/
AsmL	AsmL is the Abstract State Machine Language for .NET	http://research.microsoft.com/fse/asml/
CAML	Microsoft Research F#	http://research.microsoft.com/projects/ilx/fsharp.htm
Cobol	Fujitsu Cobol	http://www.adtools.com/info/whitepaper/net.html
Forth	Delta Forth	http://www.dataman.ro/dforth/
Eiffel	Interactive Software Engineering Eiffel for .NET	http://www.eiffel.com/products/envsn10/
Fortran	Lahey/Fujitsu Fortran for .NET	http://www.lahey.com/dotnet.htm
Fortran	Salford FTN95 for Microsoft .NET	http://www.salfordsoftware.co.uk/compilers/ftn95/dotnet.shtml
Haskell	Hugs98 for .NET	http://galois.com/~sof/hugs98.net/
Mercury	Melbourne University Mercury Project	http://www.cs.mu.oz.au/research/mercury/dotnet.html
ML	Standard ML from Microsoft Research	http://www.research.microsoft.com/Projects/SML.NET/index.htm
Mondrian	Mondrian for .NET	http://www.mondrian-script.org/
Oberon	ETH Active Oberon for .net	http://www.oberon.ethz.ch/lightning/
Pascal	Queensland University Component Pascal	http://www.citi.qut.edu.au/research/plas/projects/cp_files/ComponentPascal.html
Perl	ActiveState Perl Dev Kit with PerlNET	http://aspn.activestate.com/ASPN/NET/
Python	ActiveState Python for .NET Research	http://starship.python.net/crew/mhammond/dotnet
Scheme	Northwestern University Hotdog Scheme	http://rover.cs.nwu.edu/~scheme/
Smalltalk	SmallScript from SmallScript LLC	http://www.smallscript.net/

.NET-a programirali, on će na kraju biti preveden u MSIL koji je uvijek jednako brz, iz kojeg god jezika bio preveden. Također, zahvaljujući zajedničkim temeljima, u jednom je modulu moguće koristiti druge module pisane u bilo kojem jeziku.

Drugim riječima, izbor jezika ovisit će ponajprije o vašim vlastitim navikama, potrebama i željama. Ili o autorima knjige koju upravo čitate. Ipak, da vas ne bismo ostavili u neznanju, evo nekoliko crtica o jezicima *po defaultu* podržanim u Visual Studiju .NET.

Visual Basic .NET

Glavni adut jezika Basic (pa sukladno tome i Visual Basica .NET) je njegova čitljivost. Gdje god je moguće naredbe nisu kratice, specijalni znakovi ili akronimi, već obične engleske riječi. Zahvaljujući tome, pukim čitanjem naredbi možete shvatiti o čemu se radi i što bi koji redak mogao značiti. Čitljivost i jasnoća čine ga idealnim jezikom za učenje, što direktno utječe na njegovu popularnost i rasprostranjenost.

Projekcije su da polovica programera koji rade na platformi .NET za to koriste jezik Visual Basic .NET. Osim jednostavnosti, takav rezultat VB.NET može zahvaliti i velikom broju ljudi koji su dosad koristili njegove starije inačice, pa im se prilikom prelaska na novu platformu činilo najlogičnije ostati pri svom jeziku.

Takva odluka je očekivana, no tek djelomično utemeljena. Naime, razlika između "starog" Visual Basica i ovoga označenog nastavkom .NET je velika – mnoge su stvari dodane ili promijenjene kako bi mogao zadovoljiti stroga pravila i zahtjeve koje svim jezicima nalaže specifikacija .NET-a. Microsoft u svojoj dokumentaciji ide toliko daleko da napominje kako se radi o različitim, no srodnim jezicima.

Iako se programerima u "starom" Visual Basicu moglo spočitnuti mnogo toga što im je bilo nedostižno u odnosu na druge jezike, s Visual Basicom .NET ta se razlika osjetno smanjila, ako ne i potpuno nestala. Ostaju, naime, još neke sitnice na koje utječe sama sintaksa, no sjetite se da .NET u svojoj arhitekturi ne favorizira niti jedan od dostupnih jezika – u svima je moguće napraviti jednako dobre i brze aplikacije.

C# .NET

C# (čita se *C-sharp*) pozicionira se kao moderan i inovativan, u potpunosti objektno orijentiran jezik koji balansira između jednostavnosti Visual Basica i snage jezika C++. Ipak, kao što i samo ime govori, radi se o jeziku koji se bazira na sintaksi C-a, tako da će programeri s iskustvom u C++-u, Javi i srodnim jezicima vrlo brzo savladati ovaj potpuno nov jezik.

Iako ga je Microsoft izmislio, C# nije isključivo Microsoftov jezik. Dvije najveće standardizacijske agencije, europska ECMA i međunarodni ISO, proglasile su C# i neke druge tehnologije iz .NET-a standardom, pa već sada taj jezik možemo pronaći u razvojnim alatima drugih proizvođača.

Ne treba ignorirati činjenicu da je C# stvoren upravo za .NET Framework. Iako Microsoft stalno naglašava da nema niti će biti favoriziranja C#-a u odnosu na ostale jezike, činjenica je da se upravo on koristi u internim projektima. Ipak, nemojte zbog toga misliti da će ostali jezici trenutno podržani u Visual Studiu .NET ikada biti dovedeni u neravnopravan položaj – jedan od najjačih aduta .NET platforme jest višejezičnost i od njega Microsoft vjerojatno nikada neće odustati.

J# .NET

Kako bi privukao što veći broj programera u Javi, nastao je jezik nazvan J#. Iako bi se svaki programer u Javi vrlo brzo snašao i u C#-u, postojanje ovog jezika omogućava puno jednostavniju migraciju postojećih aplikacija i iskorištavanje postojećeg znanja. J# je Microsoftov adut i u akademskim krugovima koji su u velikom broju prihvatili Javu.

Treba imati na umu da, unatoč izuzetnoj sličnosti, J# nije Java, nema veze s platformom Java Virtual Machine i može se koristiti isključivo za razvoj aplikacija unutar okruženja .NET.

Iako za njega postoji tek djelomična podrška u Visual Studiju, postoji i jezik nazvan JScript.NET. Radi se o nadogradnji jezika JScript prilagođenoj okruženju .NET-a koja, iako podudarna u nekim stvarima, nije isto što i J#.

Visual C++ .NET

Probajte nekom programeru u C++ reći da njegov jezik može ravnopravno zamijeniti neki tamo C# – dobit ćete takvu bujicu rečenica da će vam biti žao što ste uopće išta govorili. Što je najgore – bit će u pravu!

Naime, u priči o C++ .NET morate zaboraviti na postulat o ravnopravnosti svih jezika jer on, osim mogućnosti dostupnih svakom jeziku, omogućava zaobilaženje .NET Frameworka (pisanje tzv. neupravljanog kôda) i rad na "stari način" – direktno s operativnim sustavom. Riječ "stari" u ovom slučaju nije negativna – ma koliko okruženje poput .NET Frameworka pružalo dodatnih mogućnosti i olakšavalo razvoj stvari koje je predvidio, teško da može zadovoljiti sve potrebe. To se posebno odnosi na pisanje specifičnih aplikacija poput *drivera* za hardverske uređaje, raznih sistemskih alata ili aplikacija kojima je krucijalna izuzetna brzina.

Štoviše, čak i klasične Windows-aplikacije zasada nisu preporučljive za pisanje u okruženju .NET-a. Naime, da bi ih korisnik mogao pokrenuti, mora imati instaliran .NET Framework, što je zahtjev koji ne ispunjava velik broj osobnih računala. Situacija će biti drugačija tek kada .NET Framework postane standardni dio svakog Windows operativnog sustava.

Pogledamo li stvari iz drugog kuta, Visual C++ .NET zapravo je ostao isti onaj Visual C++ koji smo poznavali još prije ere .NET-a. Dolazak .NET-a proširio mu je mogućnosti u obliku C++ Managed Extensions, biblioteke koja omogućava korištenje svega što .NET Framework nudi. Sukladno tome, C++ .NET može poslužiti za postepenu migraciju postojećih aplikacija na platformu .NET.

Karakteristike, prednosti i mogućnosti Visual C++ .NET-a (i jezika C++ općenito) izlaze iz okvira ove knjige, pa se njima nećemo baviti.

Odlučite li koristiti C++ isključivo za pisanje upravljanih aplikacija, radit ćete u podskupu ovog jezika koji se naziva Managed C++. On je po mogućnostima jednak i karakteristikama usporediv sa svakim drugim upravljanim jezikom u .NET-u. Tom programeru, onome koji radi isključivo u Managed C++, možete slobodno reći da mu jezik nije ništa bolji od C#.

Vaš prvi program

Kako se proučavanje sintakse što slijedi ne bi svelo na teoretsko razglabanje, prije ćemo naučiti jednostavan način kako možete sami isprobati svaki primjer i eksperimentirati mijenjajući ga.

Za taj posao poslužit će nam kompajleri – alati koji će prevoditi kôd koji napišemo u izvršnu datoteku s nastavkom ".exe" koju ćemo moći pokrenuti i na ekranu vidjeti rezultat. Nemojte očekivati čuda – u ovom poglavlju zadržat ćemo se na tzv. konzolskim aplikacijama dok prave poslastice čuvamo za kasnija poglavlja.

Za početak, napišimo jednostavan program...

Zdravo, svijete!

Na početku učenja bilo kojeg jezika tradicionalno se poseže za komadom kôda popularno nazvanu "Hello, world". Radi se o programčiću kojem je jedini zadatak ispisati na ekranu izraz "Zdravo, svijete!", no već iz njega možete vizualno doživjeti jezik i primijetiti osnovnu strukturu programa.

VB.NET

```
Imports System
Public Module ZdravoSvijete
 Sub Main()
      Console.WriteLine ("Zdravo, svijete!")
   End Sub
End Module
```

C#

```
using System;
class ZdravoSvijete
{
   static void Main()
   {
      Console.WriteLine ("Zdravo, svijete!");
   }
}
```

U primjerima što slijede nećemo ponavljati kostur kôda već samo dio koji obrađujemo. Vama na dušu stavljamo da, ukoliko kôd želite isprobati u praksi, ne zaboravite ga umetnuti u blok ovog primjera na kojem se nalazi linija koja počinje s Console.WriteLine.

Kompajliranje

Prepišite jedan od primjera u datoteku, snimite je pod imenom koje vam se sviđa u jednu od mapa na računalu. Preporuka je da kôd pisan u C#-u snimate s nastavkom ".cs" (primjerice, "zdravosvijete.cs"), a datotekama s kôdom u VB.NET-u dajete nastavak ".vb".

Pronađite na disku mapu u kojoj se nalaze datoteke "csc.exe" i "vbc.exe". Radi se o mapi koja se nalazi unutar sistemske mape "Windows" (odnosno "WinNT", ovisno o verziji operativnog sustava), zatim "Microsoft.NET", pa "Framework" te na kraju mapa istog imena kao i inačica .NET Frameworka koji imate instaliran. U našem slučaju je to verzija 1.1.4322, pa cijela putanja glasi:

```
C:\WINDOWS\Microsoft.NET\Framework\v1.1.4322
```

Zatim pokrenite komandnu liniju (engl. *Command Prompt*) i upišite sljedeću naredbu (ne zaboravite promijeniti putanju do mape, ukoliko se ona na vašem računalu razlikuje od naše:

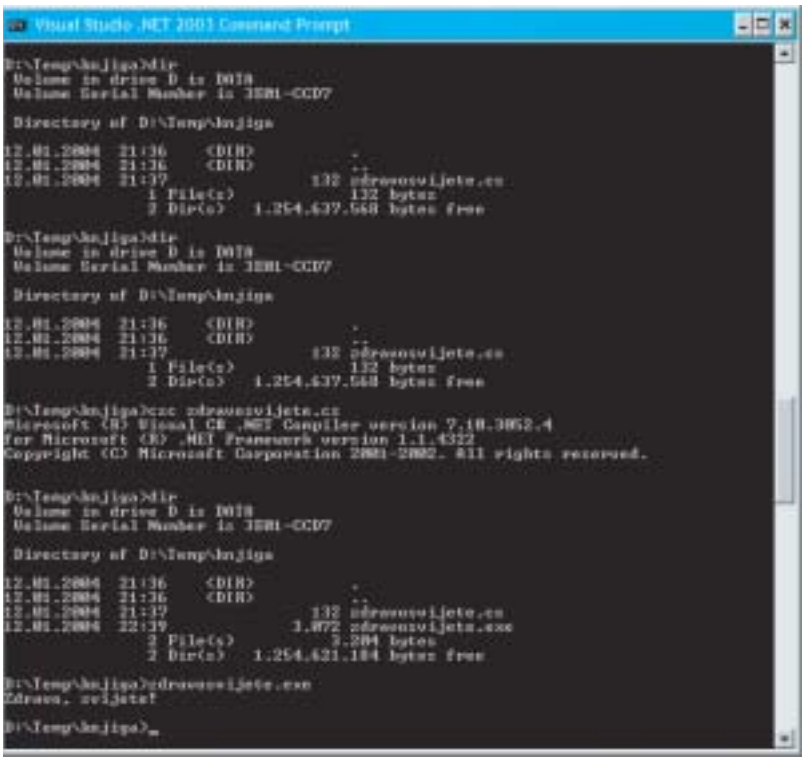
```
path "C:\WINDOWS\Microsoft.NET\Framework\v1.1.4322",%PATH%
```

Nakon toga trebate pomoću komandne linije doći do mape u koju ste smjestili datoteku s kôdom. Pretpostavljamo da osnovne naredbe DOS-a za snalaženje po mapama znate, no ipak evo nekoliko natuknica za čitatelje mlađe generacije.

Ukoliko imate instaliran Visual Studio .NET, ne morate se mučiti s traženjem mape s kompajlerima i njezine registracije naredbom "path". Dovoljno je u izborniku "Start" potražiti stavku "Visual Studio .NET 2003 Command Prompt".

Prvo treba doći na disk na kojem se nalazi mapa, a zatim upisati i samu putanju do mape. Primjerice, ukoliko je putanja mape "D:\My Documents\Primjeri", do nje ćete doći na sljedeći način:

```
d:
cd "\My Documents\Primjeri"
```



Slika 3-1: ***Postupak kompajliranja datoteke "zdravosvijete.cs" u "zdravosvijete.exe" i pokretanje te male aplikacije***

Nakon ovih srednjovjekovnih muka, na red dolazi uživancija – kompajliranje. Da biste kompajlirali datoteku "zdravosvijete.cs" u kojoj je kôd pisan u C#-u, upišite sljedeće:

```
csc zdravosvijete.cs
```

Slična je procedura i ako ste se odlučili za kôd u VB.NET-u:

```
vbc zdravosvijete.vb
```

U oba slučaja, nakon nekoliko trenutaka, u mapi će se pojaviti datoteka "zdravosvijete.exe". Ona je rezultat kompajliranja datoteke s kôdom; da biste je izvršili, dovoljno ju je pokrenuti. Rezultat će biti linija s natpisom "Zdravo, svijete!". Vaš prvi program radi!

Kad smo savladali kompajliranje aplikacija možemo se baciti na proučavanje sintakse odabranih jezika. Zapamtite da svaki od primjera što slijede možete isprobati na gore opisani način: snimite kôd u datoteku, kompajlirajte je i pokrenite dobivenu izvršnu datoteku. I, ne zaboravite na najzabavniji dio – nakon što se uvjerite da primjer radi (odnosno da ste ga dobro prepisali), probajte ga malo izmijeniti i proučavajte kako se onda ponaša.

Sintaksa

Kao što ljudski jezik ima svoj pravopis i gramatiku, tako i računalni jezici imaju pravila pisanja koje možemo ujediniti pod pojmom sintakse. Sintaksa je, jednostavno rečeno, način na koji morate pisati naredbe u nekom programskom jeziku da bi ih kompajler mogao razumjeti i pravilno protumačiti.

Svi programski jezici imaju slične mogućnosti, no različit način na koji se te mogućnosti koriste. To je posebno naglašeno u okruženju .NET-a gdje je isti kôd pisan u različitim jezicima sličniji nego ikada (zbog istih biblioteka klasa koje koriste), pa se razlike među njima svode na razlike u sintaksi.

Iako to sintaksa niti jednog jezika ne zahtijeva, preporučujemo vam da kôd pišete uredno i dosljedno – uvlačite redove, ne ostavljajte nepotrebne razmake i pišite komentare kako biste se kasnije u njemu lakše snalazili.

Kako bi poznavatelji C-olike sintakse lakše razumjeli VB-ovu i obrnuto, kroz primjere pisane u Visual Basicu .NET i C#-u prolazit ćemo paralelno. Naravno, slobodni ste jedan od jezika u potpunosti ignorirati i usvajati saznanja samo o onome koji vam se više dopada, no voljni smo se kladiti da ćete se kad-tad vratiti i pročitati preskočeno.

Velika i mala slova

Jedna od osnovnih razlika između jezika VB.NET i C# jest tretiranje malih i velikih slova. Tradicionalno, VB.NET ne pravi razliku, pa je sasvim svejedno na koji ćete od sljedećih načina pisati kôd:

VB.NET

```
Console.WriteLine ("Zdravo, svijete!")
Console.WRITELINE ("Zdravo, svijete!")
console.writeline ("Zdravo, svijete!")
CoNsOlE.WrItElIne ("Zdravo, svijete!")
```

Ipak, radi urednosti i preglednosti, preporučujemo vam da kôd pokušavate pisati što dosljednije i urednije, po mogućnosti prema pravilima koja vrijede u drugim jezicima jer će vam to olakšati čitanje i prevođenje.

C#

U C# jeziku od krucijalne je važnosti koristite li mala ili velika slova. Za razliku od VB-a .NET, kompajler će prepoznati samo prvi redak, dok će za sve ostale prijaviti grešku:

```
Console.WriteLine ("Zdravo, svijete!");
Console.WRITELINE ("Zdravo, svijete!");
console.writeline ("Zdravo, svijete!");
CoNsOlE.WrItElIne ("Zdravo, svijete!");
```

Kraj naredbe

Prošla dva primjera razlikuju se samo u znaku točka-zarez. Kao što vidimo, svaka naredba u jeziku C# mora završavati tim znakom, dok je u VB-u .NET dovoljno prijeći u novi red. Mogli bismo reći da C# ignorira prelazak u novi red i da on služi isključivo ljudima kako bi kôd bio pregledniji.

Evo primjera kako napisati tri naredbe u dva reda. Kod primjera pisanog u VB-u .NET uočite da, ukoliko u istom redu želimo staviti dvije naredbe, moramo koristiti znak dvotočke.

VB.NET

```
Console.WriteLine ("Prva linija!")
Console.WriteLine ("Druga linija!") : Console.WriteLine ("Treća linija!")
```

C#

```
Console.WriteLine ("Prva linija!");
Console.WriteLine ("Druga linija!"); Console.WriteLine ("Treća linija!");
```

Komentari

Iako bismo ih teško mogli nazvati najvažnijima, zgodno je da odmah na početku naučimo kako se u kôd upisuju komentari jer ćemo ih u primjerima što slijede i sami koristiti. Radi se o izrazima koje kompajler ignorira, a služe isključivo programerima kako bi se lakše snalazili u svom i tuđem kodu.

VB.NET

U VB-u .NET komentare označavamo znakom apostrofa ili naredbom "rem" (skraćeno od *remark*). Jednom korištena oznaka "komentira" sve napisano do kraja reda. Ova dva retka su sinonimi:

```
' Ovo je komentar koji kompajler ignorira
Rem Ovo je komentar koji kompajler ignorira
```

Oznaka komentara ne mora biti korištena na početku retka. Dozvoljeno je da u prvom dijelu bude neka naredba, a nakon nje komentar:

```
Console.WriteLine ("Testiranje komentara!") ' Komentar
```

Iako je logično, mnogi se ne sjete da su komentari zapravo zgodan način da neke naredbe privremeno, iz bilo kojeg razloga, ne budu izvršene. Primjerice:

```
Console.WriteLine ("Testiranje komentara!")
' Console.WriteLine ("Ova linija neće biti izvršena!")
```

C#

Za C# vrijede sve varijacije kao i kod VB-a .NET. Razlika je u tome što se umjesto znaka apostrofa koriste dvije kose crte:

```
// Ovo je komentar koji kompajler ignorira
```

Osim toga, C# pruža mogućnost komentiranja više linija. U VB-u .NET svaka bi linija trebala na početku imati poseban znak apostrofa, dok u C#-u to izgleda ovako:

```
/*
Ovo je komentar
koji se proteže
kroz nekoliko linija
*/
```

Varijable i njihovo deklariranje

Varijable su svojevrsni spremnici u koje spremate određene vrijednosti. Kada vam ta vrijednost kasnije u kodu zatreba, jednostavno upišete ime varijable i dobit ćete jednak efekt kao da ste upisali njenu vrijednost. Evo jednostavnog primjera u kojem prvi i drugi red imaju isti efekt:

VB.NET

```
Console.WriteLine (5)
broj = 5 : Console.WriteLine (broj)
```

C#

```
Console.WriteLine (5);
broj = 5; Console.WriteLine (broj);
```

Međutim, da biste neku varijablu mogli koristiti, morate je deklarirati. Postupak deklaracije nije ništa drugo nego način da kažete kompajleru da ćete koristiti neku varijablu te da joj odredite koji tip vrijednosti će moći poprimiti. O samim varijablama i tipovima više ćete saznati nešto kasnije, u petom poglavlju, a ovdje nam je zadatak naučiti sintaksu za deklariranje varijabli. Stoga ćemo u primjerima koristiti najpoznatije tipove – *string* (znakovni niz), *integer* (cijeli broj) i *object* (o tome nešto kasnije – spominjemo ga radi specifične sintakse).

Varijable možete imenovati gotovo proizvoljno. Kažemo "gotovo" jer postoje određena pravila kojih se treba držati. Najvažnije je da ime varijable mora počinjati slovom, ne smije sadržavati razmake i razne druge specijalne znakove (među koje ubrajamo i "naša slova") te ne smije biti istog imena kao ključne riječi, naredbe i slično. Treba imati na umu da C# pazi i na način na koji ste napisali ime varijable, tako da varijable "broj" i "Broj" neće biti iste. VB.NET ne pravi razliku među različito napisanim imenima varijabli.

Varijable možete imenovati i samo jednim slovom (kao što mi najčešće radimo u primjerima), no u većim programima to neće biti praktično (ako ništa drugo, neće biti dovoljno slova). Zato vam preporučujemo da varijable pišete u obliku riječi, tako da vas njihovo ime automatski asocira na vrijednost koju sadrži. Evo nekoliko zgodnih primjera:

```
ImeDatoteke
Ime_i_Prezime
UkupnaVrijednostNarudzbe
DatumZaposlenja
```

VB.NET

Glavna razlika između naša dva jezika na području varijabli jest što VB.NET podržava tzv. implicitno deklariranje varijabli odnosno, razumljivijim rječnikom, moguće je isključiti obavezno deklariranje varijabli prije korištenja (što se, kad se već razbacujemo pojmovima, zove eksplicitno deklariranje). Iako vam to nikako ne bismo preporučili, to možete učiniti tako da na početku datoteke s kôdom napišete:

```
Option Explicit Off
```

Nadajući se da to ipak nećete učiniti (i tako si na prvi pogled olakšati, no dugoročno zakomplicirati programerski život), evo kako se deklariraju varijable u VB-u .NET:

```
Dim n As Integer
Dim s As String
Dim s1, s2 As String
Dim objekt1         ' ukoliko se ne navede, podrazumijeva se tip Object
Dim objekt2 As New Object()
```

C#

U C#-u deklariranje istih tih varijabli izgledalo bi ovako:

```
int n;
String s;
String s1, s2;
Object objekt1;
Object objekt2 = new Object();
```

Prilikom deklariranja moguće je varijablama odmah dodijeliti neke početne vrijednosti. Naravno, ništa vas ne sprečava da te početne vrijednosti dodijelite nekoliko linija kasnije, no na ovaj način svoj kôd činite preglednijim, a dobivate i na brzini. Dakako, ponekad vam istovremeno deklariranje i pridruživanje vrijednosti neće biti pogodno, no ako vam zatreba, evo kako ga postići:

VB.NET

```
Dim s As String = "Zdravo, svijete!"
Dim n As Integer = 1
```

C#

```
string s = "Zdravo, svijete!";
int n = 1;
```

Odluke

Uzmimo za primjer da želimo napisati programčić koji će, ovisno o vrijednosti neke varijable, ispisati na ekranu neki tekst. Za to nam treba provjera vrijednosti te varijable, što radimo naredbom *if*. Ta je naredba prisutna u svim programskim jezicima, pa ako imate imalo programerskog iskustva znate čemu ona služi. (Ako slučajno nemate, shvatit ćete iz primjera.)

U primjerima što slijede prvo ćemo deklarirati varijablu, dodijeliti joj vrijednost te nakon toga, ako je vrijednost veća od 100, ispisati adekvatan tekst. (Uočite da je uvjet u C# **uvijek** u zagradama, dok uvjet u VB-u .NET može, ali ne mora biti.)

VB.NET

```
Dim n As Integer
n = 123
If n > 100 Then Console.WriteLine("Broj je veći od sto!")
```

C#

```
int n;
n = 123;
if (n > 100) Console.WriteLine("Broj je veći od sto!");
```

U praksi su rijetki slučajevi u kojima postoji samo jedna naredba koju želite izvršiti u slučaju zadovoljenja uvjeta. Najčešće je potrebno napraviti više stvari, poput ispisivanja dvaju redaka teksta kao u našem sljedećem primjeru:

VB.NET

```
' izostavljena je deklaracija varijabli
If n > 100 Then
   Console.WriteLine("Broj je veći od sto!")
   Console.WriteLine("Hvala što ste koristili program.")
End If
```

C#

```
// izostavljena je deklaracija varijabli
if (n > 100)
```

```
{
   Console.WriteLine("Broj je veći od sto!");
   Console.WriteLine("Hvala što ste koristili program.");
}
```

U gornjim će primjerima ako uvjet nije ispunjen program ignorirati naredbe u *if*-bloku. Međutim, često postoji potreba da i u tom slučaju izvršite neku naredbu; primjerice, ispišete da je broj manji od sto. Evo kako:

VB.NET

```
If n > 100 Then
   Console.WriteLine("Broj je veći od sto!")
Else
   Console.WriteLine("Broj je manji od sto!")
End If
```

C#

```
if (n > 100)
{
   Console.WriteLine("Broj je veći od sto!");
}
else
{
   Console.WriteLine("Broj je manji od sto!");
}
```

Naravno, niti ovo ne mora zadovoljiti sve vaše potrebe. U sljedećim primjerima imat ćemo tri različita natpisa: jedan za brojeve do sto, drugi za one između sto i dvjesto, te treći za sve ostale.

VB.NET

```
If n > 200 Then
   Console.WriteLine("Broj je veći od dvjesto!")
ElseIf n > 100 Then
   Console.WriteLine("Broj je između sto i dvjesto!")
Else
   Console.WriteLine("Broj je manji od sto!")
End If
```

C#

```
if (n > 200)
{
   Console.WriteLine("Broj je veći od dvjesto!");
}
else if (n > 100)
{
   Console.WriteLine("Broj je između sto i dvjesto!");
}
else
{
   Console.WriteLine("Broj je manji od sto!");
}
```

Primjećujete i sami – kada bi postojao velik broj *else*-blokova, kôd bi postao nezgrapan i nepregledan. Zato u pomoć pozivamo naredbu *case*. Slijedi kôd iste funkcionalnosti kao i prošli primjer:

VB.NET

```
Select Case n
   Case Is > 200
      Console.WriteLine("Broj je veći od dvjesto!")
   Case 100 To 200
      Console.WriteLine("Broj je između sto i dvjesto!")
   Case Else
      Console.WriteLine("Broj je manji od sto!")
End Select
```

C#

Očekujete primjer u C#? Nažalost, ovakav tip uvjeta u tom je jeziku nemoguć. Moguće je samo navođenje konkretnih vrijednosti varijable, no ne i raspona. Dakle, pomoću naredbe *case* u C#-u možemo napisati samo provjeru je li varijabla jednaka određenoj vrijednosti, dok ćemo za raspone morati koristiti naredbu *if*.

```
switch (n) {
   case 200 :
      Console.WriteLine("Broj je jednak dvjesto!");
      break;
```

```
    case 100 :
        Console.WriteLine("Broj je jednak sto!");
        break;
    case 0 :
        Console.WriteLine("Broj je jednak nuli!");
        break;
    default :
        // ukoliko niti jedan uvjet nije zadovoljen, izvršit će se blok "default"
        Console.WriteLine("Radi se o nekom drugom broju!");
        break;
}
```

Unatoč tom prilično velikom ograničenju, C# nudi neke mogućnosti koje VB.NET ne poznaje. Pogađate i sami – ključ je u naredbi *break,* koju može zamijeniti izrazima poput *goto case 0*. Napišemo li nešto poput ovoga, izvršit će se i zadovoljeni blok i blok na koji skačemo.

```
switch (n) {
    case 200 :
        Console.WriteLine("Broj je jednak dvjesto!");
        goto case 0; // skok u blok "case 0"
    case 100 :
        Console.WriteLine("Broj je jednak sto!");
        goto default; // skok u blok "default"
    case 0 :
        Console.WriteLine("Broj je jednak nuli!");
        break;
    default :
        Console.WriteLine("Radi se o nekom drugom broju!");
        break;
}
```

Uvjeti

U priči o odlukama koristili smo uvjete na temelju kojih bi naredbe *if* i *case* donosile odluku o izvršavanju jednog od blokova kôda. Mi smo u primjerima koristili najjednostavnije ("veće od", "manje od") koje možemo naći i u udžbenicima iz matematike u osnovnoj školi.

Vrijeme je da upoznamo i naprednije oblike uvjeta. Svima će biti zajedničko da daju logičku vrijednost "istina" ili "laž". Pogledajmo sljedeći uvjet:

```
n > 100
```

Ukoliko je vrijednost varijable *n* veća od 100, rezultat će biti istina (engl. *true*), a ako je vrijednost varijable manja (ili jednaka) broju 100, dobit ćemo logičku vrijednost neistine (engl. *false*). Usporedni prikaz osnovnih operatora uspoređivanja možete vidjeti u tablici 3-1.

VB.NET	C#	Opis
=	==	jednako
<	<	manje od
<=	<=	manje od ili jednako
>=	>=	veće ili jednako
>	>	veće
<>	!=	nije jednako (različito)

Tablica 3-1: ***Usporedba osnovnih operatora uspoređivanja***

Složene uvjete možemo slagati i kombiniranjem više jednostavnih. Pokušajte odgonetnuti što znače sljedeći uvjeti. (Imajte na umu da, za razliku od klasične matematike, uvijek koristimo zaobljene zagrade, u koliko god nivoa postojale).

VB.NET

```
(n > 100) AND (n < 200)
(n < 100) OR (n > 200)
(n > 100) OR ((n > 0) AND (n < 50))
NOT (n > 100)
```

C#

```
((n > 100) && (n < 200))
((n < 100) || (n > 200))
((n > 100) || ((n > 0) && (n < 50)))
(!(n > 100)
```

(Mala digresija: sjetite se kada smo pričali o VB-u .NET i njegovoj glavnoj prednosti – čitljivosti napisanog kôda. Upravo u gornjim primjerima ona dolazi do izražaja!)

Nakon što smo odgonetnuli ove jednostavnije primjere, možemo se upoznati i s nekim naprednijim logičkim operatorima (vidi tablicu 3-2).

Tablica 3-2:
Usporedba osnovnih logičkih operatora

VB.NET	C#	Rezultat je istinit ako...
AND	&&	...su obje vrijednosti istinite
OR	||	...je barem jedna vrijednost istinita
XOR	^	...je samo jedna vrijednost istinita
NOT	!	...negira vrijednost izraza koji slijedi

Kao što vidimo u primjerima, uvjeti u C#-u uvijek moraju biti omeđeni zagradama, dok u VB-u .NET to nije slučaj. Zagradama određujemo, baš kao u pučkoškolskoj matematici, i redoslijed računanja vrijednosti. Uočite kako sljedeća dva retka ne daju identične rezultate (a usput i kako drugi redak zapravo nema smisla, no ostavismo ga kako bismo naglasili ulogu zagrada):

VB.NET

```
(n > 100) OR ((n > 0) AND (n < 50))
((n > 100) OR (n > 0)) AND (n < 50)
```

C#

```
((n > 100) || ((n > 0) && (n < 50)))
(((n > 100) || (n > 0)) && (n < 50))
```

Spomenimo još i da rezultat, logičku vrijednost nekog uvjeta možete spremiti u varijablu. Taj tip varijable naziva se *boolean*. Evo kako to izgleda u praksi:

VB.NET

```
Dim b1, b2 As Boolean
b1 = True
b2 = n > 100
```

C#

```
bool b1, b2;
b1 = true;
b2 = (n > 100);
```

Sve napisano u ovom odlomku tek je vrh ledene sante. Uvjeti su mnogo kompleksnijih od ovih nekoliko operacija, no ograničen prostor ne dozvoljava nam dublji ulazak u problematiku. Neke druge polovit ćete u hodu kroz primjere koji slijede, a utješno je što ćete u većini slučajeva koristiti upravo ove, najjednostavnije oblike.

Računske operacije

Iako je to računalu trivijalan zadatak, možete ga zadužiti za najjednostavnije računske operacije kao što su zbrajanje, oduzimanje, množenje i dijeljenje. U tablici 3-3 možete vidjeti kako pojedine računske operacije izgledaju u pojedinom jeziku

VB.NET	C#	Opis
10 + 50	10 + 50	zbrajanje, rezultat je 40
200 – 100	200 – 100	oduzimanje, rezultat je 100
20 * 10	20 * 10	množenje, rezultat je 200
432 / 10	432 / 10	dijeljenje, rezultat je 43,2
432 \ 10	(nije direktno dostupno)	dijeljenje s cjelobrojnim rezultatom, rezultat je 43
432 mod 10	432 % 10	ostatak cjelobrojnog dijeljenja, rezultat je 2

Tablica 3-3: ***Usporedba osnovnih računskih operacija***

S dobivenim rezultatima možete raditi svašta, uključujući i dodjeljivanje varijablama. Primjerice, kod pridruživanja vrijednosti nekoj varijabli, tu vrijednost ne morate eksplicitno napisati; ona može biti dana u obliku neke računske operacije. Od jednostavnih operacija slobodno možete slagati složene, sa zagradama ili bez njih. Osim brojeva, u izrazima možete koristiti i varijable. Evo nekoliko primjera:

VB.NET

```
n = 10 + 50          ' n = 40
m = 200 – 100        ' m = 100
n = m / 10           ' n = 100 / 10 = 10
n = n + 10           ' n = 10 + 10 = 20
m = m / (n - 10)     ' m = 100 / (20 – 10) = 100 / 10 = 10
```

C#

```
n = 10 + 50;          // n = 40
m = 200 – 100;        // m = 100
n = m / 10;           // n = 100 / 10 = 10
n = n + 10;           // n = 10 + 10 = 20
m = m / (n - 10);     // m = 100 / (20 – 10) = 100 / 10 = 10
```

U računskoj operaciji dijeljenja vrlo je važno kojeg je tipa varijabla kojoj pridružujemo vrijednost. Naime, dok cjelobrojnoj varijabli nećete niti moći eksplicitno dodijeliti decimalni broj, ako se dogodi da rezultat vašeg dijeljenja bude decimalni broj, a varijabla kojoj ga želite pridružiti cjelobrojnog tipa, rezultat će automatski biti pretvoren u cijeli broj.

Neke od računskih operacija često se koriste. Najčešći slučaj je povećanje vrijednosti varijable za jednu jedinicu (recimo: n = n + 1). Kako bi programeri bili produktivniji, sintakse jezika dozvoljavaju skraćeno pisanje nekih češće korištenih izraza (tablica 3-4).

Tablica 3-4: ***Skraćeni oblici računskih operacija; umjesto broja 5 može stajati bilo koji broj ili varijabla***

VB.NET	C#	Dugi oblik
n += 5	n += 5	n = n + 5
n –= 5	n –= 5	n = n – 5
n *= 5	n *= 5	n = n * 5
n /= 5	n /= 5	n = n / 5
(nema posebnog izraza)	n ++	n = n + 1
(nema posebnog izraza)	n ––	n = n - 1

C#

Jezik C# časti nas još jednom vrlo praktičnom poslasticom zvanom uvjetno pridruživanje. Da ne kompliciramo teorijom, objasnit ćemo vam primjerom. Sljedeća dva retka daju identičan rezultat (varijabla n cjelobrojnog je tipa, dok je varijabla s *string*):

```
if (n > 0) s = "n je pozitivan"; else s = "n je negativan";
s = n > 0 ? "n je pozitivan" : "n je negativan";
```

Ukoliko je uvjet (u našem slučaju n > 0) zadovoljen, varijabli s bit će dodijeljena vrijednost "n je pozitivan". Ako taj uvjet nije zadovoljen, s će biti jednak izrazu "n je negativan".

Računsku operaciju zbrajanja, osim nad brojevima, možete vršiti i nad tekstom (*stringovima*). Radi se o najobičnijem spajanju dvaju (ili više) stringova u jedan. To se "zbrajanje *stringova*" zove konkatenacija, a u praksi izgleda ovako:

```
str = "Mali " + "zeko";     // str = "Mali zeko"
str += " i prijatelji";     // str = " Mali zeko i prijatelji"
```

Isto je moguće i VB.NET-u – samo izbacite točka-zarez i zamijenite oznaku za komentar. U tom se jeziku, osim operatora "+" u ovu namjenu koristi i operator "&".

Petlje

Petlje služe za ponavljanje nekog dijela kôda određeni broj puta ili dok se neki uvjet ne ispuni. U svakodnevnom životu primjera za to ima koliko hoćete. Ako u dućanu znate točan broj komada nekog proizvoda koji želite kupiti, istu ćete radnju uzimanja i spremanja u košaricu ponoviti točno određen broj puta. Kada, pak, dođete na blagajnu, iz novčanika ćete vaditi novac sve dok ne izvadite iznos veći ili jednak potrošenom.

Za početak ćemo se zabaviti prvim slučajem – ponavljanjem neke radnje određen broj puta. Za to će nam poslužiti petlja *for*. U sljedećem primjeru na ekran ćemo ispisati brojeve od 1 do 100:

VB.NET

Sintaksa u VB-u .NET je jednostavna – nakon naredbe *for* odredite varijablu koju želite "vrtiti" u petlji te nakon znaka jednakosti odredite raspon kojim će se ona kretati. Prilikom svakog izvršavanja bloka naredbi između riječi *for* i *next*, odabrana varijabla će imati drugu vrijednost. Prvi put jedan, zatim dva, pa tri... i tako sve do sto.

```
Dim n As Integer ' ne zaboravite prije deklarirati varijablu!
For n = 1 To 100
   Console.WriteLine(n)
Next
```

C#

Ova sintaksa na prvi je pogled znatno složenija, no zato pomoću nje, kao što ćemo vidjeti kasnije, možemo postići puno više varijacija. Primijetite da se nakon naredbe *for* u zagradi navode tri parametra odijeljena točka-zarezom. U prvom određujemo koja će se varijabla koristiti kao brojač i njenu početnu vrijednost. U drugom se parametru stavlja uvjet koji treba biti ispunjen da bi se blok naredbi izvršavao. Treći parametar služi najčešće za uvećavanje brojača (što se u VB-u .NET ne mora definirati, već je *defaultna* vrijednost). U našem ga slučaju uvećavamo za jedan, pa pišemo izraz tome namijenjen (vidi dio teksta o računskim operacijama).

```
for (int n = 1; n <= 100; n++) // varijablu možete deklarirati i ovdje
{
   Console.WriteLine(n);
}
```

For-petlja nije ograničena isključivo na povećavanje vrijednosti za jedan. U primjerima što slijede pokazat ćemo kako napisati petlju kojoj će brojač povećavati vrijednost za dva, u kojoj će vrijednost brojača padati i, konačno, gdje će se brojač povećavati eksponencionalno.

VB.NET

```
For n = 1 To 100 Step 2 ' vrijednosti 1, 3, 5, 7, 9 .. 97, 99
  ...
Next

For n = 100 To 1 Step -1 ' vrijednosti 100, 99, 98 .. 3, 2, 1
  ...
Next

' zahvaljujući dosjetkama u bloku, vrijednosti će biti 1, 2, 4, 8, 16, 32, 64
For n = 1 To 100
   ...
   n = n * 2   ' ili n *= 2
   n = n - 1   ' ili n -= 1
Next
```

C#

```
for (int n = 1; n <= 100; n+=2) // vrijednosti 1, 3, 5, 7, 9 .. 97, 99
   { ... }
```

```
for (int n = 100; n >= 1; n–) // vrijednosti 100, 99, 98 .. 3, 2, 1
   { ... }

for (int n = 1; n <= 100; n*=2) // vrijednosti 1, 2, 4, 8, 16, 32, 64
   { ... }
```

Osim *for*-petlje, postoji i oblik petlje nazvan *while*. Osnovna razlika među ovim petljama jest što potonja nema brojač – promjenu uvjeta koji odlučuje o ponavljanju petlje morat ćete sami kontrolirati u bloku unutar petlje. Ovaj tip petlje najčešće se koristi kada unaprijed ne znamo koliko puta će se neka petlja trebati izvršiti.

Ipak, i petlji *while* možemo dodati brojač te tako dobiti isti efekt kao i s petljom *for.* Evo primjera:

VB.NET

```
Dim n As Integer
n = 1
Do While n <= 100
  Console.WriteLine(n)
  n += 1
Loop
```

C#

```
int n = 1;
while (n <= 100) {
  Console.WriteLine(n);
  n += 1;
}
```

Primjećujete i sami da je petlja *for* u ovakvim slučajevima jednostavnija za korištenje. Prava vrijednost petlje *while* očitovat će se u sljedećem primjeru. U njega uvodimo jednu novu naredbu – *Console.ReadLine*. Dok kod naredbe *Console.WriteLine* određeni izraz ispisujemo na ekran, nova će naredba tražiti od korisnika programa da sam nešto upiše. Sljedeća petlja izvršavat će se sve dok korisnik ne upiše riječ “izlaz”.

VB.NET

```
Dim unos As String = ""
Do While unos <> "izlaz"
```

```
    Console.WriteLine("Unesite tajnu riječ:")
    unos = Console.ReadLine()
Loop
```

C#

```
string unos = "";
while (unos != "izlaz")
{
  Console.WriteLine("Unesite tajnu riječ:");
  unos = Console.ReadLine();
}
```

Metode ili funkcije

Uhvatite li se prilikom programiranja da neki komad kôda često ponavljate, mogli biste si uštedjeti puno vremena kreiranjem funkcije. Nakon što određeni komad kôda smjestite u funkciju, bit će dovoljno pozvati njeno ime i taj će kôd biti izvršen. U sljedećem primjeru funkcija se brine o ispisivanju izraza “Pozdrav svima!”.

VB.NET

```
Imports System
Public Module ZdravoSvijete

   Sub IspisiPozdrav()
      Console.WriteLine("Pozdrav svima!")
   End Sub

   Sub Main()
      IspisiPozdrav()
      IspisiPozdrav()
      IspisiPozdrav()
   End Sub

End Module
```

C#

```
using System;
class ZdravoSvijete
```

```
{
    static void IspisiPozdrav()
    {
        Console.WriteLine("Pozdrav svima!");
    }

    static void Main()
    {
        IspisiPozdrav();
        IspisiPozdrav();
        IspisiPozdrav();
    }
}
```

Sigurno ste primijetili da prvi put u nekom primjeru ponovo koristimo kostur iz primjera "Zdravo, svijete!". Kao što smo tada napomenuli, sav kôd treba ići u blok Main(), za koji sada znamo da je zapravo funkcija, osnovna funkcija koja se izvršava po pokretanju programa.

Kostur pokazujemo zato što deklariranje funkcija ne može biti unutar neke druge funkcije, već isključivo izvan. Ono što možemo (i moramo) raditi unutar neke druge funkcije jest pozivanje, što mi u primjeru radimo unutar funkcije Main(). U primjerima što slijede ponovno ćemo, radi uštede prostora, izostavljati kostur kôda, a vi imajte na umu da se deklariranje funkcija radi izvan, a pozivanje funkcija unutar glavne funkcije.

Pozivanje funkcija moguće je vršiti i iz drugih funkcija, a ne samo osnovne. Takvih ćemo primjera imati u kasnijim poglavljima. Također, pojedinu je funkciju moguće pozvati i iz nje same. Te se pak funkcije nazivaju rekurzivnima i u radu s njima treba biti vrlo oprezan. I njih ćemo susresti nešto kasnije u knjizi.

Gornji primjeri u praksi nemaju nekog smisla – više bismo profitirali da smo triput zaredom napisali naredbu Console.WriteLine s pripadajućim parametrom. Međutim, zamislite da pozivi te funkcije nisu jedan ispod drugoga, već razbacani po programu, a vama dođe zadatak da tekst koji biva ispisan izmijenite u "Dobar dan!". Puno je jednostavnije izmijeniti stvar na jednom mjestu, nego tražiti gdje se sve spomenuta fraza koristi i mijenjati je na svim tim mjestima.

Još bolji primjer korisnosti funkcija je slučaj u kojem kôd unutar funkcije nije, kao u našem primjeru, samo jedna linija, nego neki složeniji skup naredbi. U takvim ćete slučajevima uštedjeti i na samom tipkanju, a i kôd koji producirate bit će puno pregledniji.

Funkcije možemo pozivati i s parametrima. U sljedećem primjeru ćemo funkciji kao parametre poslati dva broja, a ona će imati zadatak oduzeti veći broj od manjeg i rezultat ispisati na ekranu.

VB.NET

```
Sub Oduzmi(broj1 As Integer, broj2 As Integer)
   If broj1 > broj2 Then
      Console.WriteLine(broj1 - broj2)
   Else
      Console.WriteLine(broj2 - broj1)
   End If
End Sub
```

C#

```
static void Oduzmi(int broj1, int broj2)
{
   if (broj1 > broj2)
   {
      Console.WriteLine(broj1 - broj2);
   }
   else
   {
      Console.WriteLine(broj2 - broj1);
   }
}
```

Pozivanje ove funkcije izgledat će ovako:

VB.NET

```
Oduzmi(10, 20)   ' rezultat 10
Oduzmi(5, 11)    ' rezultat 6
```

C#

```
Oduzmi(123, 321);  // rezultat 198
Oduzmi(100, 100);  // rezultat 0
```

Funkcije možemo nazivati i njihovim drugim imenom – metode – a funkcije koje smo dosad upoznali možemo zvati i procedurama. Za procedure je specifično da ne vraćaju nikakvu povratnu vrijednost, dok one koje ne pripadaju pod značenje tog pojma, vraćaju vrijednosti. Te vraćene vrijednosti možemo spremiti u varijablu, uvrstiti u računski izraz ili pak iskoristiti kao parametar za neku drugu metodu.

U sljedećem primjeru modificirat ćemo prethodnu funkciju tako da rezultat oduzimanja ne ispisuje na ekran, nego vraća u obliku svoje povratne vrijednosti.

VB.NET

```
Function Oduzmi(ByVal broj1 As Integer, ByVal broj2 As Integer) As Integer
   If broj1 > broj2 Then
      Return broj1 - broj2
   Else
      Return broj2 - broj1
   End If
End Function
```

C#

```
static int oduzmi(int broj1, int broj2)
{
   if (broj1 > broj2)
   {
      return broj1 - broj2;
   }
   else
   {
      return broj2 - broj1;
   }
}
```

Funkcija koja vraća vrijednost ima dva obilježja koja ne nalazimo u ostalim funkcijama. U redu deklariranja funkcije dodan je parametar koji označava tip vrijednosti koji će funkcija vraćati. Sintaksa za to vrlo je slična deklaraciji varijabli – u VB-u .NET koristi se riječ "As", nakon koje slijedi tip vrijednosti (u našem slučaju "Integer"), dok se u C#-u tip vrijednosti stavlja na mjesto riječi "void" (hrv. praznina; u našem slučaju "int").

Drugo obilježje je definiranje povratne vrijednosti. Za taj posao koristimo riječ "return", nakon koje navodimo vrijednost koju će funkcija vratiti pozivatelju (u našem slučaju rezultat izraza "broj1 – broj2" odnosno "broj2 – broj1").

Uz ove dvije, Visual Basic pravi još jednu razliku među funkcijama koje vraćaju i ne vraćaju vrijednosti. One koje ne vraćaju, procedure, započinju naredbom "Sub", a završavaju s "End Sub". Ove druge pak na početku imaju "Function", a na kraju "End Function".

I ovakvu funkciju možete pozvati na isti način kao i prošle, samo što će onda njezina povratna vrijednost biti izgubljena. Kako to u većini slučajeva nema smisla, u sljedećem ćemo primjeru pokazati neke od načina na koje možete tu vrijednost "uhvatiti" i iskoristiti.

VB.NET

```
Console.WriteLine(Oduzmi(90, 50))    ' vraćena vrijednost je ispisana
Dim n As Integer = Oduzmi(400, 500) ' vraćena vrijednost je pridružena varijabli n
n = Oduzmi(Oduzmi(100, 200), Oduzmi(500, 600))
```

C#

```
Console.WriteLine(Oduzmi(90, 50)); ' vraćena vrijednost je ispisana
int n = Oduzmi(400, 500);          ' vraćena vrijednost je pridružena varijabli n
n = Oduzmi(Oduzmi(100, 200), Oduzmi(500, 600));
```

Klase

Sada kada razumijemo pojam, možemo reći da su naredbe koje smo koristili u primjerima (WriteLine i ReadLine) zapravo metode. Naravno, njih nismo morali sami definirati jer postoje u osnovnoj biblioteci klasa, i to u klasi Console. Zato prilikom pozivanja navodimo *namespace* klase u kojoj se metoda nalazi, kako bi kompajler znao gdje da je nađe.

Čak štoviše, da mu eksplicitno ne kažemo, kompajler ne bi znao niti gdje da traži određenu klasu. Zato na početku kôda pišemo "Imports System" odnosno "using System", ovisno o jeziku. To je direktiva kompajleru da sve vanjske klase na koje se pozivamo potraži u baznoj klasi koja ima *namespace* "System".

I sami primjeri koje smo koristili sami po sebi zapravo predstavljaju klasu. Naravno, da bi oni funkcionirali na način na koji to rade bazne klase, potrebne su određene adaptacije, posebno u primjeru pisanom u VB-u .NET, koji poznaje drugačiju sintaksu za nešto zvano standardni modul koji smo u primjerima koristili. No o svemu tome više u poglavlju posvećenom objektno-orijentiranom programiranju...

Zbogom, vizualni bejziče...

Naše druženje s Visual Basicom .NET ovdje prestaje iako se nesumnjivo o Visual Basicu .NET još štošta može reći. Međutim, kao što smo već naglašavali, fokus knjige bit će na jeziku C#, pa će odsada svi primjeri biti isključivo u tom jeziku.

Ipak, vjerujemo da ćete se i dalje vraćati na ovo poglavlje. Ono je, naime, osim jednostavnijeg svladavanja novog jezika osobama s programerskim iskustvom u Basicu, zamišljeno i kao vodič za razumijevanje primjera u VB-u .NET na koje možete naići. Dogodit će se, naime, da nećete moći naći rješenje za problem koji vas muči pisano u C#-u. Zahvaljujući usporednim primjerima u ovom poglavlju, moći ćete lakše odgonetnuti o čemu se radi i prevesti primjer kako bi se uklopio u vašu aplikaciju.

4. POGLAVLJE

Razvojna okolina

U ovom poglavlju:

- Što je Visual Studio .NET
- Kako se snaći u sučelju
- Stvaranje projekata
- Ključna olakšanja u radu
- Korištenje dokumentacije (MSDN)

Pisanje iole veće aplikacije onako kako smo to radili u prošlom poglavlju bilo bi, pogađate, srednjovjekovno mučenje. Čak su i u davna vremena tekstualnih operativnih sustava postojali programi kojima je bio zadatak olakšati pisanje kôda i njegovo kompajliranje. Zahvaljujući marketingu, danas se takvi programi nazivaju "integrirana razvojna okolina" (engl. *Integrated Development Environment*), no njihova bit i dalje ostaje ista – pojednostaviti pisanje kôda.

Razvojni alati postoje za sve operativne sustave i platforme, a vrijednost pojedinog alata očituje se u njegovim mogućnostima. Upravo prema tim mogućnostima programeri (odnosno, ako ćemo koristiti marketinške pojmove, *developeri*) biraju razvojnu okolinu u kojoj će raditi.

Ponajbolji izbor za razvoj aplikacija u okruženju .NET-a jest Microsoftov Visual Studio .NET iako postoje i drugi kvalitetni razvojni alati, poput Borlandovog C# Buildera.

Visual Studio .NET

Već niz godina Microsoftova razvojna okolina zove se Visual Studio. Prije prelaska u vode .NET-a zadnja verzija označena je brojkom 6. Nakon što je predstavljen .NET, na tržište je izašla nova verzija pod nazivom Visual Studio .NET, a u trenutku pisanja ove knjige aktualna je ona punog imena Visual Studio .NET 2003. Svaka od inačica Visual Studija .NET prati određeno izdanje .NET Frameworka. Tako je Visual Studio .NET (ponekad referenciran kao VS 7.0) bio pratitelj inačice 1.0, a Visual Studio .NET 2003 (poznat i kao VS 7.1) .NET Frameworka verzije 1.1. U razvoju su i sljedeće inačice .NET Frameworka 2.0 i pratećeg mu Visual Studija .NET kodnog imena Whidbey koje bi trebale donijeti niz poboljšanja i prednosti. Zanimljivo je da postoje planovi i za inačicu nakon Whidbeyja, nazvanu Visual Studio Orcas, u kojoj će biti poraðeno na tješnjoj integraciji s novom inačicom Windowsa kodnog imena Longhorn.

Da smo se ranije sreli...

Microsoft je oduvijek svoje platforme volio obogaćivati razvojnim alatima. Štoviše, dio zasluga za popularizaciju Microsoftovih proizvoda pripisuje se izrazitoj brizi za programere. Tako smo još od davne 1981. u MS-DOS-u mogli raditi u programskom jeziku Basic, a s pojavom Windowsa 1990. godine izdana je i prva inačica vizualnog razvojnog alata nazvanog Visual Basic. Sljedeći veliki korak napravio je Visual Studio izdan 1995. godine, u kojem je omogućen razvoj internetskih rješenja, no nisu zapostavljene niti klasične aplikacije za Windowse.

Visual Studio .NET odnosno cijela .NET inicijativa zapravo je unifikacija programskih modela – povezivanje dotada nepovezanih metoda i principa u konzistentno programsko sučelje neovisno o jeziku ili tipu aplikacije.

Ukoliko ste se izgubili u netom pročitanoj salvi podataka, ne brinite – sve što trebate znati jest da ćemo se u pregledu primarno orijentirati na posljednju finalnu inačicu – Visual Studio .NET 2003.

Visual Studio .NET 2003 dostupan je u četiri izdanja, namijenjena različitim profilima korisnika. Osnovna inačica naziva se Professional i sadrži sve mogućnosti namijenjene individualnom razvoju aplikacija. Nedostaju tek neke napredne mogućnosti modeliranja i korištenja baza podataka, upravljanja kôdom na kojem istovremeno radi više korisnika te ne sadrži razvojne licence za korištenje Microsoftovih serverskih proizvoda.

Izdanje Academic namijenjeno je isključivo obrazovnim institucijama, a osim svih karakteristika izdanja Professional može se pohvaliti posebnim edukativnim alatima za korištenje u nastavi, kao i puno većom bazom primjera i dokumentacije.

Naprednijim programerima trebat će izdanja Enterprise Developer odnosno Enterprise Architect koja donose nekoliko ključnih mogućnosti za razvoj velikih rješenja. Kako njihove karakteristike prelaze gabarite ove knjige, nećemo ih eksplicitno navoditi, no sve zainteresirane upućujemo na adresu http://msdn.microsoft.com/vstudio/howtobuy/choosing.aspx gdje će naći detaljnu usporednu tablicu.

Procesor	Pentium II 450 MHz (preporučeno Pentium III 600 MHz)
Operativni sustav	Windows Server 2003
	Windows XP Professional
	Windows XP Home Edition (ne podržava ASP.NET)
	Windows 2000 Professional SP3
	Windows 2000 Server SP3
Radna memorija	160 MB (Windows Server 2003, Windows XP Professional)
	96 MB (Windows XP Home Edition, Windows 2000 Professional)
	192 MB (Windows 2000 Server)
Tvrdi disk	900 MB na sistemskom disku
	3,3 GB na instalacijskom disku
	1,9 GB za dokumentaciju (MSDN Library)

Tablica 4-1: Minimalni sistemski zahtjevi za pokretanje Visual Studija .NET 2003

Instalacija Visual Studija

Četiri su koraka instalacije Visual Studija .NET. Prvi ili – bolje rečeno – nulti korak je instalacija zakrpa i dodataka nužnih da bi Visual Studio uopće mogao raditi. Svi se potencijalno potrebni dodaci nalaze na instalacijskom DVD-u (odnosno CD-u "Microsoft Visual Studio .NET 2003 Prerequisites"), pa nema potrebe za instalacijom preko Interneta. Oko detekcije potrebe za obaveznim dodacima pobrinut će se sam instalacijski program, tako da ćete nakon ovog koraka biti bogatiji za nekoliko Service Packova, Microsoft FrontPage 2000 Web Extensions Client i sam Microsoft .NET Framework 1.1. Ukoliko to odaberete, instalirat će se i Microsoft Visual J# .NET Redistributable

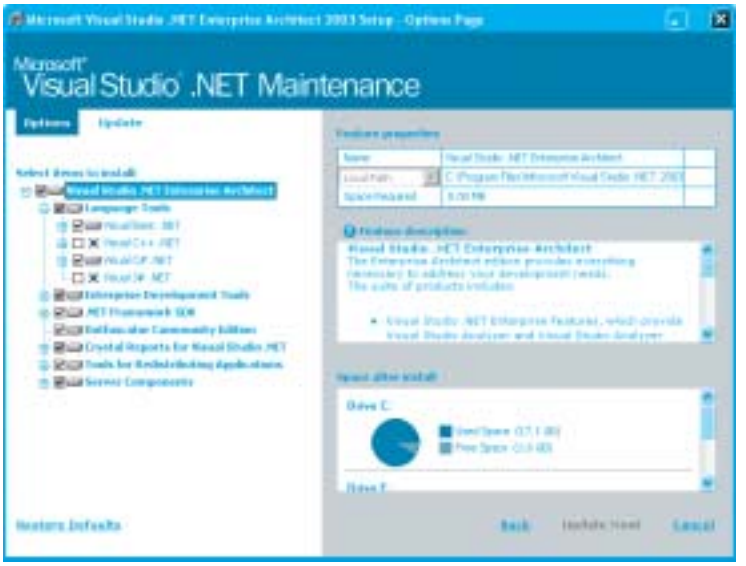
Package 1.1. Međutim, taj potonji "paket" vam treba samo ako planirate razvijati u jeziku J#. Ako vam to nije namjera, pravovremeno ga isključite kako ne bi zauzimao dragocjen diskovni prostor.

Slika 4-1:
Instalacija Visual Studija sastoji se od četiri koraka.

Na računalu može istovremeno biti instalirano više različitih inačica Visual Studija.

Sljedeći korak je instalacija samog Visual Studija. Vjerujemo da sa snalaženjem u sučelju nećete imati problema, no savjetujemo vam da malo više vremena potrošite na pregledavanje komponenti koje dolaze u paketu. Svaka od komponenti sadrži detaljan opis, pa prema njemu možete barem otprilike zaključiti što nećete koristiti, te na temelju toga izdvojiti iz instalacije. Taj savjet posebno vrijedi za odabir programskih jezika – gotovo je sigurno da nećete koristiti sva četiri ponuđena.

Ukoliko se prvi put susrećete s programiranjem ili nemate iskustva u jezicima Java i C++, najsigurniji odabir su Visual C# .NET i Visual Basic .NET (kao što je prikazano na slici 4-2).



Slika 4-2: Probajte isključiti mogućnosti koje sigurno ne namjeravate koristiti – Visual Studio je glomazan komad softvera, pa nema smisla da neke njegove komponente samo troše diskovni prostor.

Ne strahujte da ćete krivo odabrati odnosno isključiti nešto što vam može kasnije zatrebati. Sve je opcije moguće naknadno dodati, ali i oduzeti.

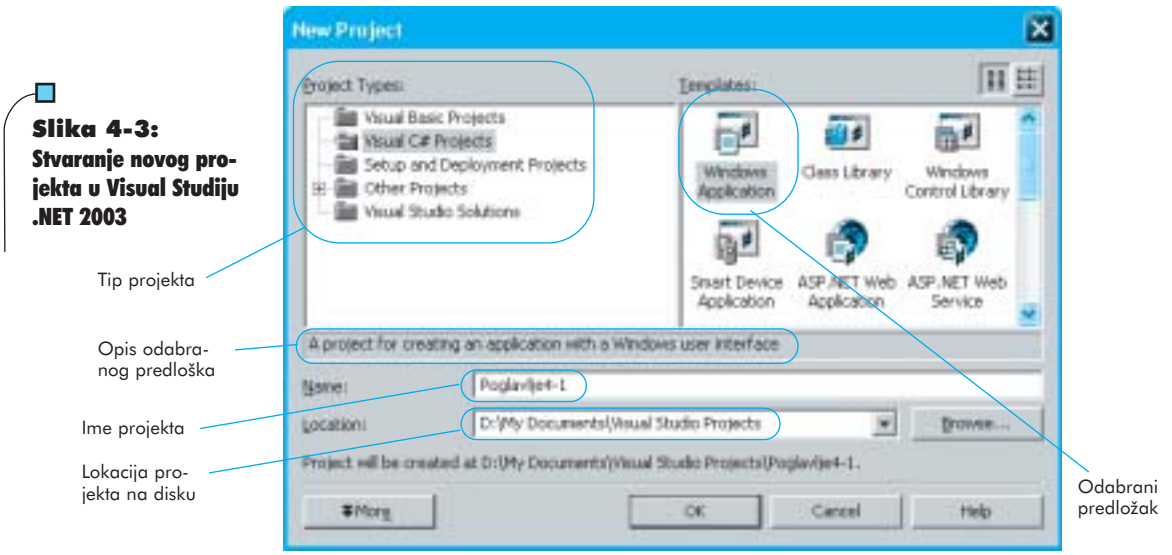
Slijedi instalacija dokumentacije – ogromne baze znanja poznate pod imenom Microsoft Developer Network (skraćeno MSDN) Library. Bez dokumentacije život programera je osjetno teži, pa svakako preporučujemo da je instalirate. Ipak, ukoliko su vam gotovo dva gigabajta diskovnog prostora prevelik danak, imajte na umu da je kompletan MSDN Library javno objavljen na Internetu, na adresi http://msdn.microsoft.com/library/. Čak štoviše, ta je inačica stopostotno ažurna i sadrži sve najnovije tekstove i specifikacije, za razliku od one na instalacijskim medijima koja je najvjerojatnije stara barem nekoliko mjeseci, ako ne i više. S druge strane, i instalirana inačica ima nekoliko prednosti, od kojih je najvažnija mogućnost zvana *dynamic help*. To u praksi znači da za vrijeme rada možete u bilo kojem trenutku stisnuti tipku F1 i dobiti informacije o dijelu sučelja ili naredbi na kojoj radite. Detaljnije informacije o MSDN Libraryju pronaći ćete na kraju ovog poglavlja.

Odmah prelazimo na zadnji, četvrti korak instalacije. Radi se o podsjetniku da je dobro s Interneta preuzeti i instalirati eventualne zakrpe, nadogradnje i *service packove* za programe koje ste upravo instalirali: imat ćete sve najnovije mogućnosti i svoje ćete računalo učiniti sigurnijim. Ne treba posebno naglašavati da ovaj korak nije nužan, no svakako je preporučljiv.

Stvaranje novog projekta

Nakon pokretanja Visual Studija dočekat će vas početna stranica (engl. *Start Page*). Na njoj ćete pronaći listu zadnje otvaranih projekata, a ispod te liste nalaze se dva ključna gumba za kreiranje

novih i otvaranje postojećih projekata koji nisu u popisu. Ukoliko vam je ovo prvo pokretanje Visual Studija, stvaranje novog projekta jedina vam je mogućnost.



Slika 4-3: Stvaranje novog projekta u Visual Studiju .NET 2003

Pri kreiranju novog projekta morate paziti na dvije ključne stvari. Prva i najvažnija je tip projekta koji želite stvoriti. Radite li u C#-u, izabrat ćete stavku "Visual C# Projects", a na desnoj strani će se pojaviti nekoliko predložaka za projekte u tom programskom jeziku. Među ostalim, možete odabrati predložak za pisanje prozorske, konzolske ili web-aplikacije. Primijetite da koristimo pojam "predložak" – svaki tip aplikacije moguće je pisati od nule (izborom predloška "Empty Project"), ne koristeći kôd koji Visual Studio automatski generira izborom nekog predloška. Međutim, takav je pristup u većini slučajeva gubljenje vremena.

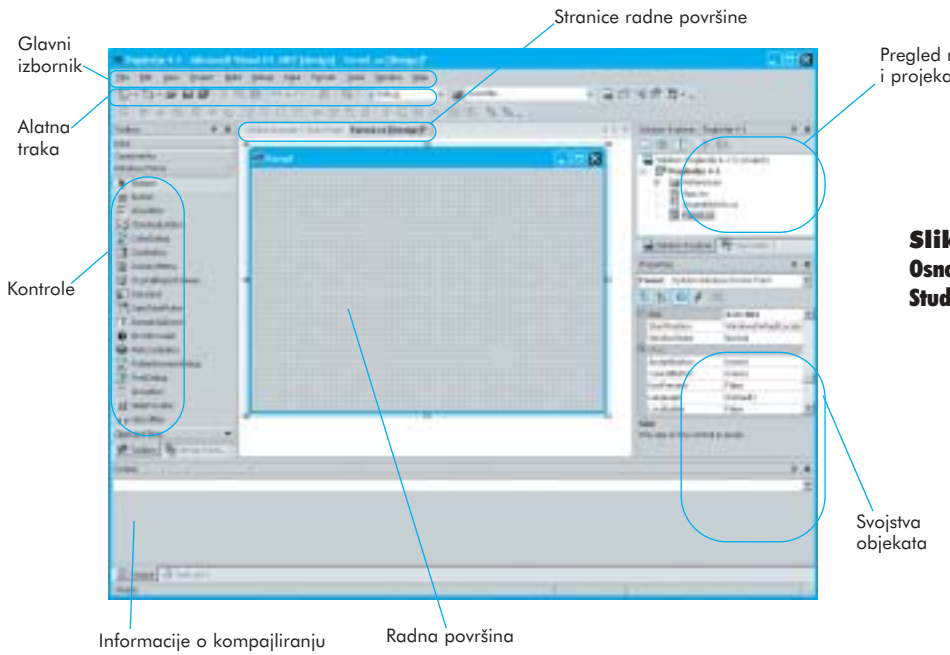
Dalje trebate paziti na ime projekta. Ime koje ovdje unesete koristit će se na raznim mjestima – od imena mape u kojoj će biti smještene datoteke koje pripadaju projektu do imena *assemblyja* i *namespacea*. Dakako, na mnogim ćete mjestima naziv moći promijeniti naknadno, no za to potreban trud i vrijeme sugeriraju da već na ovom koraku razmislite i upišete nešto pametno i logično.

Za imenovanje projekata odnosno aplikacija ne postoje stroga pravila, no praksa je da projekt dobije ime koje opisuje njegovu funkcionalnost. Naravno, ne treba pretjerivati u dužini i preporučljivo je izbjegavati znakove poput našem jeziku specifičnih palatala te razmake.

Treba spomenuti da je pojam projekta adekvatan pojmu *assemblyja* o kojem smo pričali u drugom poglavlju. Spajanjem dvaju ili više projekata (*assemblyja*) stvaramo rješenje. Prema inicijalnim postavkama, prilikom stvaranja novog projekta otvara se i novo rješenje (engl. *solution*), no zahvaljujući opcijama pri kreiranju projekta (djelomično skrivenih iza klika na "More"), takvom je ponašanju moguće doskočiti. Sasvim je izvjesno da vam za jednostavne aplikacije igranje s rješenjima neće trebati, no s obzirom na to da se taj pojam često pojavljuje, red ga je spomenuti kako vas ne bi zbunjivao.

Sučelje razvojne okoline

Mi smo za početak izabrali predložak za prozorsku aplikaciju, no prije nego što krenemo na prve redove kôda, zadržat ćemo se koji trenutak na sučelju koje nas je dočekalo nakon potvrde izbora.



Slika 4-4: Osnovno sučelje Visual Studija .NET 2003

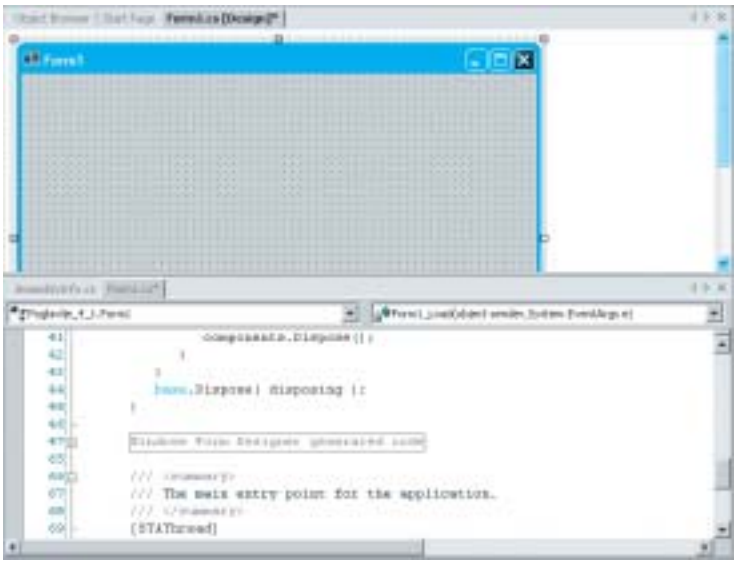
Kažu da slika vrijedi tisuću riječi, a kako imate iskustva s radom u prozorskim aplikacijama, vjerujemo da ćete brzo prepoznati dijelove koji se i ovdje pojavljuju. Ipak, ovo je sučelje znatno bogatije, zbog čega ćete vrlo brzo poželjeti veliki monitor. Naime, radnu površinu koja dominira sredinom ekrana okružuju pomoćni prozori koji su, za razliku od nekih drugih aplikacija, ne samo korisni, nego i nužni.

Štoviše, svaki od prozora krije više funkcionalnosti. Tako prozor Toolbox može postati Server Explorer, Output može poslužiti za popisivanje zadataka koji vas čekaju (Task List), Solution Explorer se zna pretvoriti u preglednik klasa (Class View), a prozor sa svojstvima brzo se transformira u dinamički sustav pomoći (Dynamic Help). No, krenimo redom – na stranicama što slijede proučit ćemo najvažnije dijelove Visual Studijeva sučelja, dok ćemo ostale upoznavati u hodu, kako nam u kojem primjeru zatrebaju.

Radna površina

U središnjem dijelu sučelja možemo otvoriti zaista sve i svašta. Sjetite se da se na tom mjestu otvorila početna stranica s popisom nedavno korištenih projekata, a nakon kreiranja projekta tu se otvorila prazna forma Windowsa (zato što smo izabrali prozorsku aplikaciju). Tu se još nalazi i Object Browser koji vam omogućava "šetanje" po *assemblyjima* i pripadajućim im *namespaceovima* i klasama, a s vremenom ćete otkriti da se u tom dijelu sučelja pojavljuju i mnoge druge funkcionalnosti.

Najčešći stanovnik ovog dijela sučelja ipak će biti datoteke s kôdom. Svaku datoteku možete otvoriti na dva načina, čak i istovremeno. Prvi način naziva se "dizajnerski način". U njemu vizualno (zato se i zove Visual Studio) kreirate formu aplikacije. Taj način prepoznat ćete po oznaci "[Design]" u imenu kartice. Kliknete li desnom tipkom miša na njega i iz padajućeg izbornika odaberete "View Code", otvorit će se nova kartica i radnu površinu zauzet će kôd koji se krije iza te forme. U kojem god načinu radili, rezultat vašeg rada bit će kôd. Jedina razlika je u tome što dok radite u dizajnerskom načinu, Visual Studio kôd generira automatski, dok ga u drugom načinu pišete sami.



Slika 4-5: Radna površina podijeljena na dva dijela, tako da istovremeno možete vidjeti i kôd i formu.

Prema *defaultnim* postavkama kôd koji se generira u dizajnerskom načinu neće biti prikazan zajedno s ostatkom koda. To ne znači da ne postoji ili da je u nekoj drugoj datoteci, nego da je skriven. Na slici 4-5 možete vidjeti pravokutnik s natpisom "Windows Form Designer generated code". Spomenuti kôd se krije iza njega, a vidjeti ga možete ako kliknete na znak plusa koji se nalazi u istom redu, lijevo od pravokutnika.

Među otvorenim karticama možete se kretati klikanjem na njihova imena, no pravo bogatstvo (posebno onima s većim monitorima) krije se iza klika desnom tipkom miša. Zahvaljujući tamo smještenim opcijama, radnu ćete površinu moći podijeliti na više dijelova i tako imati više istovremeno otvorenih kartica.

Klikanje desnom tipkom miša otkrit će i razliku između otvorenih datoteka i pomoćnih prozora kao što su Object Browser ili Start Page. Potonje, naime, možete sakriti, pretvoriti u plutajući prozor ili smjestiti na rub Visual Studijeva sučelja i tako u potpunosti prilagoditi razvojnu okolinu svojim željama i potrebama.

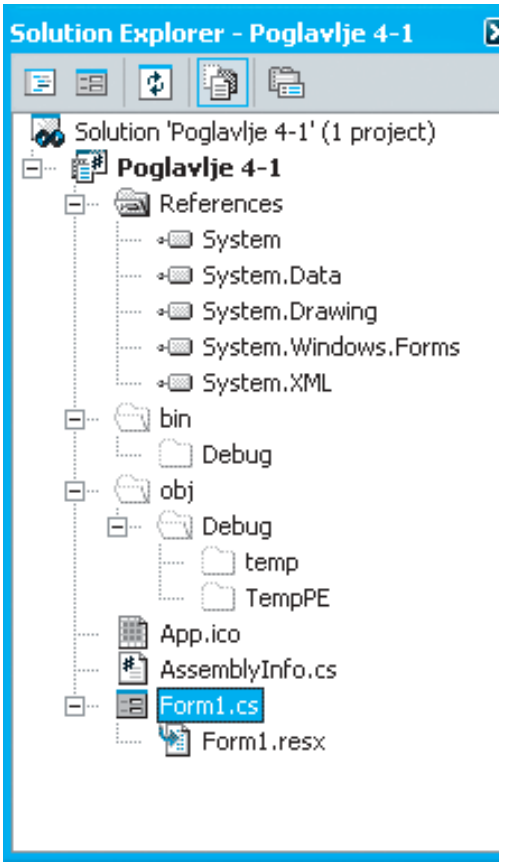
Ime izbornika	Opis
File	otvaranje projekata, rješenja i datoteka, zatvaranje istih, ispis na štampač, izlaz iz programa
Edit	radnje s međuspemnikom (*clipboard*) i pretraživanje
View	prikazivanje i uklanjanje prozora i alatnih traka te upravljanje istima
Project	upravljanje projektom, dodavanje formi, kontrola, komponenti, klasa
Build	kompajliranje programa
Debug	naredbe vezane uz *debugiranje* programa i njegovo pokretanje
Data	rad s bazama podataka
Format	naredbe za uređivanje i mijenjanje parametara kontrola na formi
Tools	razni dodatni alati i mogućnosti za prilagođavanje sučelja (sve za što nije bilo mjesta drugdje)
Windows	slaganje i upravljanje otvorenim prozorima
Help	pristup sustavu pomoći

Tablica 4-2: Kratki vodič kroz organizaciju glavnog izbornika

Solution Explorer

Jedno rješenje sastoji se od velikog broja datoteka i drugih resursa, čak i kada se radi o najjednostavnijim aplikacijama. Da vam snalaženje među njima, dodavanje novih, uklanjanje postojećih i navigacija općenito budu što jednostavniji brine se Solution Explorer.

Slika 4-6:
: Solution Explorer – tek smo stvorili rješenje, a već ima toliko resursa

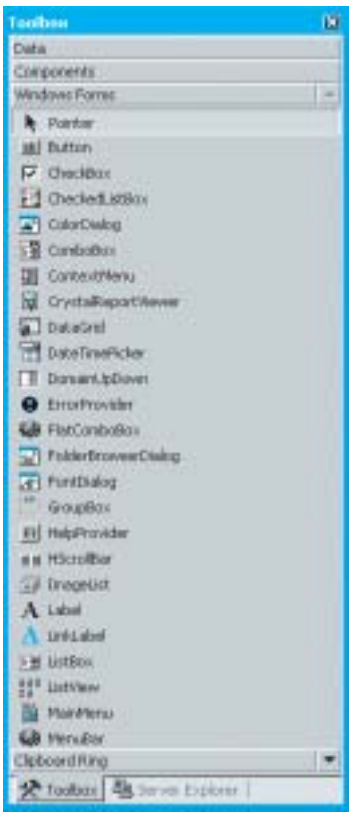


Ovisno o tipu datoteke odnosno resursa koji je označen, traka s alatima na vrhu prozora mijenjat će dostupne mogućnosti. Primjerice, ukoliko označite projekt, klikom na ikonu "Properties", dobit ćete prozor sa svojstvima projekta, a ukoliko označite neku datoteku s kôdom, moći ćete je otvoriti u kodnom ili dizajnerskom načinu. Uostalom – isprobavajte, jedino što se može dogoditi jest da si zatrpate radnu površinu prozorima, no to je lako ugasiti.

Kao što smo već spomenuli, sve pomoćne prozore, pa tako i ovaj, možete smjestiti negdje drugdje, ako vam tako bolje paše. Nismo spomenuli mogućnost "Auto Hide", koja će se posebno svidjeti vlasnicima manjih monitora. Ona pomoćne prozore svodi na usku traku uz rub, koja se, ukoliko vam funkcija iz prozora zatreba, otvara klikom miša, a nakon korištenja automatski zatvara. Mogućnost "Auto Hide" uključujete klikom na ikonu pribadače ili izborom adekvatne stavke iz padajućeg izbornika, koji se pojavljuje nakon što desnom tipkom miša kliknete na ime prozora.

Toolbox

Pomoćni prozor Toolbox svoje najkorisnije lice pokazuje u dizajnerskom načinu rada. U njemu se, naime, nalaze kontrole koje možete uzimati i odvlačiti na formu s njihove desne strane. Na taj način možete dodati tekst, kućicu za upis, padajući izbornik i još mnogo toga. Kako smo se prilikom kreiranja projekta odlučili za stvaranje prozorske aplikacije, kontrole će biti prilagođene našem izboru. Da smo, primjerice, izabrali predložak za web-aplikaciju, izbor kontrola bio bi sasvim drugačiji.



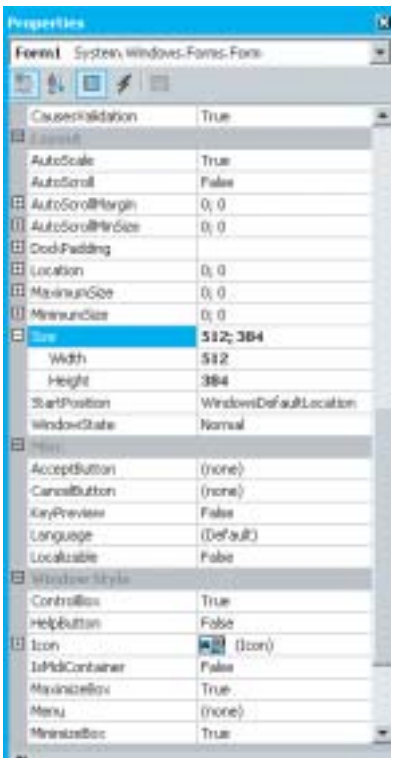
Slika 4-7:
Pomoćni prozor Toolbox predstavlja bit vizualnog programiranja.

Ovakvo vizualno programiranje izuzetno olakšava stvaranje aplikacija. Svako odvlačenje kontrole na formu rezultirat će prilično velikim komadom kôda koji, kao što smo već spomenuli, ne samo da ne morate znati napisati nego ne morate niti vidjeti.

Dostupnih kontrola ima puno, pa su zbog jednostavnijeg snalaženja poslagane u nekoliko grupa (Data, Components, Windows Forms, General...). I tu do izražaja dolazi prilagodljivost sučelja – grupe možete kreirati, brisati, sortirati i preslagivati: ako vam se ne sviđa postojeći raspored, slobodno napravite organizaciju koja vam se čini efikasnijom. Osim grupama, iste stvari možete raditi i kontrolama, no o tome neki drugi put. Ukoliko ste željni samostalnog isprobavanja, ključni prvi korak je klik desnom tipkom miša...

Svojstva (*Properties*)

Kad postavite neku kontrolu na stranicu, poželjet ćete promijeniti neka njezina svojstva. Formi ćete najvjerojatnije mijenjati veličinu, kućici za upis poziciju unutar forme, padajućem izborniku ponuđene vrijednosti... Sve je to, dakako, moguće napraviti izravnom intervencijom u kôdu, no čemu se patiti kad postoji pomoćni prozor Properties.



Slika 4-8:
Svojstva kontrola jednostavno se podešavaju u pomoćnom prozoru *Properties*.

Kao što i samo ime govori, on sadrži listu svih svojstava kontrole koja se mogu mijenjati (postoje, naime, svojstva čije vrijednosti nije moguće mijenjati). Želite li promijeniti svojstvo neke kontrole, prvo je trebate označiti, bilo mišem u dizajnerskom načinu, bilo izborom iz liste kontrola u samom prozoru Properties (vidi sliku 4-8).

Nakon toga u listi se pojavljuju svojstva specifična za odabranu kontrolu. Svojstva možemo pregledavati po abecedi (zgodno uvijek kada točno znate naziv svojstva koje tražite) ili po grupama (kada ne znate točan naziv svojstva, no znate čemu služi).

Primijetit ćete da su neki nazivi svojstava napisani u zagradi. To znači da to nisu svojstva kontrole, nego neke druge vrijednosti vezane uz nju. Najpoznatiji primjer takvog "svojstva" je identifikacijsko ime kontrole – označeno nazivom Name.

Organizacija liste svojstava je jednostavna. Na lijevoj strani su nazivi svojstava, a na desnoj pripadajuće vrijednosti. Želite li promijeniti neku vrijednost, možete je ručno upisati ili odabrati iz padajućeg izbornika, ukoliko isti postoji. Neka složenija svojstva umjesto padajućeg izbornika imaju zasebne pomoćne prozore koje otvaramo klikom na gumbić označen točkicama (s konkretnim primjerima upoznat ćemo se kasnije u poglavlju i knjizi). Vrijednosti koje promijenite bit će deblje ispisane nego vrijednosti koje niste dirali, što u praksi značajno doprinosi snalaženju među svojstvima.

Događaji (*Events*)

Osim svojstava, svaka kontrola ima i događaje. Događaji su trenuci u kojima se dogodi nekakva promjena vezana uz kontrolu. Primjerice, kada u kućicu za upis upišemo neki tekst, nastaje događaj TextChanged. Ukoliko tom događaju pridružimo neku funkciju, ta će se funkcija izvršiti svaki put kada nastane taj događaj. Prenesimo to na naš primjer – svaki put kada korisnik programa izmijeni tekst u kućici za upis, nastat će događaj TextChanged. Ako za njega vežemo funkciju, onda će ona biti izvršena svaki put kada korisnik izmijeni sadržaj kućice za upis.

Listu događaja neke kontrole možemo pronaći u istom prozoru u kojem i svojstva, samo je potrebno kliknuti na ikonu munje. Vezanje funkcije uz neki događaj je jednostavno – u popisu pronađete događaj koji želite vezati, dvokliknete na stupac lijevo od naziva događaja i Visual Studio će vam u središnjem dijelu sučelja otvoriti kôd programa i kreirati kostur funkcije. Vama jedino preostaje napisati sadržaj funkcije, no o tome nešto kasnije...

Vaš drugi program

Dosta razgledavanja, krenimo na konkretan primjer! Vjerujemo da ste korak stvaranja novog projekta već odradili te vas čeka prazna forma. Kako se inicijalno radi o formi skromnih dimenzija, prvi će nam zadatak biti njeno povećanje. To možete napraviti na dva načina. Vizualno – jednostavno razvući formu do veličine koju želite ili mijenjanjem svojstava Size u pomoćnom prozoru Properties.

Kad smo već kod svojstava, promijenit ćemo još jedno. To će biti naslov forme (ne brkajte ga s imenom kontrole!) – svojstvu Text upišite vrijednost "Naš drugi program" (bez navodnika). Primijetit ćete da se naslov prozora odmah ispravio i u dizajnerskom načinu.

Nakon toga dolazi vrijeme da u prozor smjestimo i određene kontrole iz pomoćnog prozora Toolbox. Odvucite s njega kontrole Label, PictureBox i Button kako biste kreirali inačice tih kontrola na formi. Uočite da su se kreirane inačice kontrola automatski nazvale *label1*, *pictureBox1* i *button1*.

Imena kreiranih kontrola možete promijeniti, no bitno je da to učinite (po mogućnosti) odmah nakon kreiranja. Naime, ime kontrole koristi se u kôdu svaki put kada želite nešto s njom napraviti (a u praksi je to vrlo često), pa ako ime promijenite naknadno, sve će reference na kontrolu biti pogrešne.

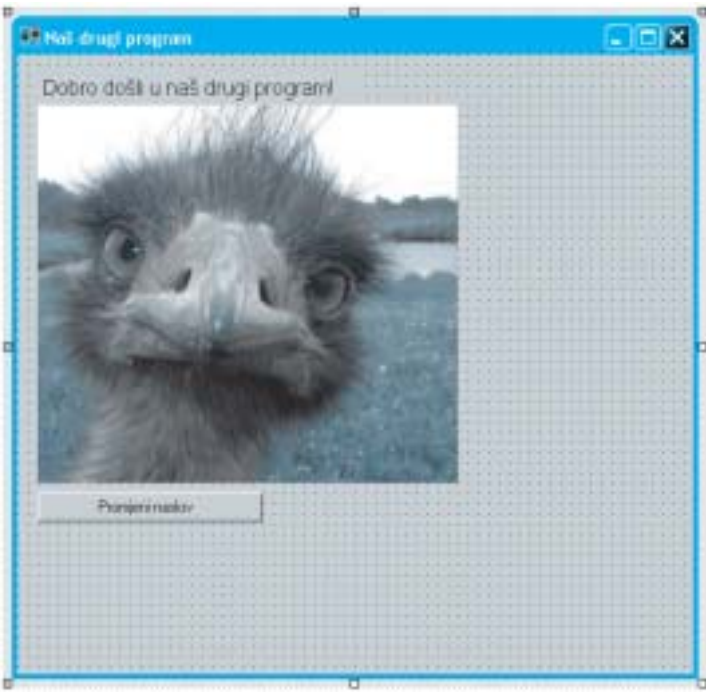
Kako bi netom postavljene kontrole imale neku ulogu, potrebno im je promijeniti svojstva. Tako ćemo prvoj kontroli (*label1*) promijeniti tekst koji prikazuje na ekranu. Umjesto postojećeg "label1" svojstvu Text upisat ćemo "Dobro došli u naš drugi program!". Kako je tekst predugačak da bi stao u jedan redak, prelomit će se u drugi, a da bismo to spriječili povećat ćemo širinu kontrole razvlačenjem. Također, pošto se radi o naslovu, promijenit ćemo veličinu slova kojom je tekst ispisan (unutar svojstva Font, redak Size).

Drugoj, slikovnoj kontroli dodat ćemo sliku. Označite je i među svojstvima potražite Image. Klikom na taj redak pojavit će se sitni gumb s tri točkice, na koji valja stisnuti da bi se otvorio prozor pomoću kojeg ćemo izabrati sliku s diska. Kako odabrana slika vrlo vjerojatno ne stane u kontrolu koju smo joj namijenili, potražite svojstvo SizeMode i postavite ga na vrijednost StretchImage. To će "natjerati" sliku da se prilagodi veličini kontrole.

Ostao nam je gumb koji ćemo zadužiti da, kada bude kliknut, promijeni natpis u prvoj kontroli. Prvo ćemo mu promijeniti natpis (svojstvo Text) u "Promijeni naslov" te ga proširiti kako bi novi natpis bio vidljiv. Nakon toga treba događaju koji nastupa kad je gumb pritisnut pridružiti funkciju koja će izmijeniti natpis prvoj kontroli. Kako se radi o *defaultnom* događaju, nije potrebno

ulaziti na listu događaja za taj objekt, već u dizajnerskom načinu treba dva puta kliknuti na gumb. Kao što smo već rekli, automatski će se generirati kostur funkcije unutar kojega treba dodati naredbu za promjenu natpisa:

```
private void button1_Click(object sender, System.EventArgs e)
{
    label1.Text = "Klik je promijenio tekst!";
}
```



Slika 4-9:
Ovako izgleda naš primjer u dizajnerskom načinu sučelja.

Treći redak našeg primjera nalaže da se svojstvu Tekst kontrole *label1* pridruži izraz "Klik je promijenio tekst!". Nemojte nam vjerovati na riječ – isprobajmo zajedno!

Sjetite se komplikacija oko kompajliranja iz prošlog poglavlja – ovdje se sve rješava pritiskom na tipku F5 ili izborom opcije Start iz izbornika Debug. Nekoliko trenutaka kasnije pokrenut će se aplikacija koju smo upravo napisali.

Kliknite na gumb "Promijeni naslov" i uočite što će se dogoditi. Kako se radi o sasvim jednostavnom programu, to je sve što ćete u njemu moći napraviti. Međutim, svrha ovog primjera nije bila velika funkcionalnost, nego da se upoznate s osnovnim principima i mogućnostima. Morate priznati, zaista je jednostavno!

Slika 4-10:
Naš primjer – pokrenut i funkcionalan

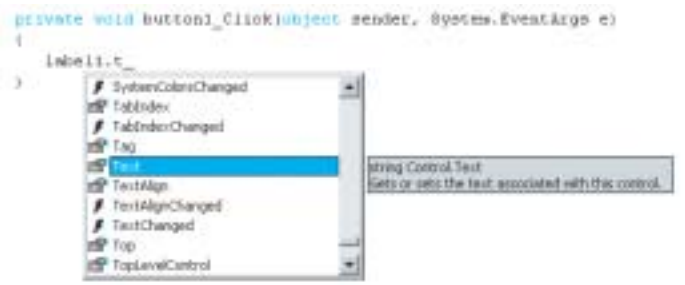
Imate li volje, možete se samostalno igrati sa svojstvima – većina njih je intuitivna i brzo ćete poloviti kako koje svojstvo utječe na kontrolu kojoj pripada. Naravno, možete i pričekati osmo poglavlje, koje je posvećeno prozorskim aplikacijama i usto puno detaljnije obrađuje kontrole i njihova svojstva.

Snalaženje u kodu

Pod ovim naslovom cilj nam je ukratko izdvojiti neke zanimljive i korisne mogućnosti Visual Studijeva sučelja vezane uz snalaženje u kôdu. Radi se o sitnicama koje se u praksi pokazuju vrlo praktičnima i uvelike olakšavaju pisanje programa.

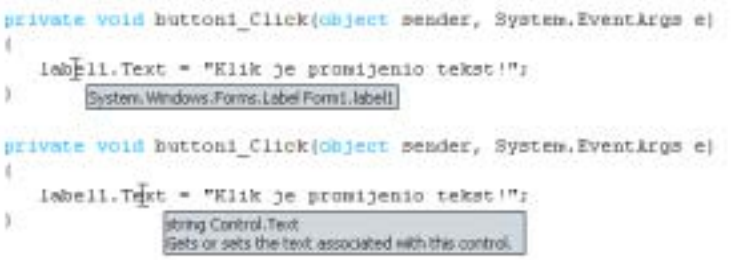
Najprije svakako valja spomenuti bojanje kôda – tako će, primjerice, u crnom tekstu ključne riječi biti ispisane plavom bojom, a komentari zelenom. Tu je i nezaobilazno poravnavanje kôda, koje će se u većini slučajeva aplicirati automatski, čak i prilikom *copy-pasteanja*. Dakako, svu je automatiku moguće isključiti, a postoji mogućnost da uvlačenje umjesto znaka *tab* ubacuje razmake (Tools > Options > Text Editor > All languages > Tabs).

Mogućnost koju ste sigurno već sreli u marketinškim materijalima pod imenom IntelliSense omogućava automatsko dopunjavanje započetih izraza. Vjerojatno ste i sami za vrijeme pisanja



Slika 4-11: IntelliSense – vjerojatno najkorisnija mogućnost Visual Studijeva sučelja

funkcije u prošlom primjeru primijetili da se, nakon što ste otipkali ime kontrole i stavili točku, otvorila padajuća lista sa svim svojstvima, metodama i inim stvarima vezanim uz tu kontrolu. Tada ste, umjesto tipkanja, jednostavno mogli izabrati željeni pojam s liste i tako brže (i, što je još važnije, točnije) napisati naredbu. Kao što možete vidjeti na slici 4-11, osim dopunjavanja započete riječi, IntelliSense nudi i kratki opis svakog pojma.

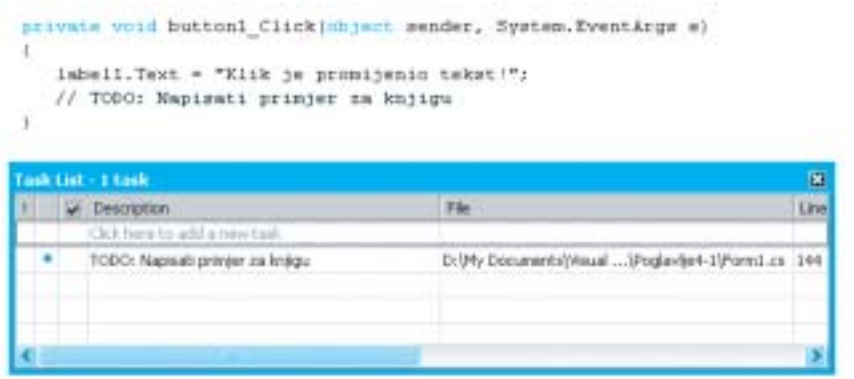


Slika 4-12: Brza identifikacija pojmova u kodu

Kratki opis pojmova na stranici nije specifičan samo za IntelliSense. Ukoliko tijekom pregledavanja kôda naiđete na nepoznat pojam, vjerojatno će pomoći ako iznad njega zaustavite pokazivač miša na nekoliko trenutaka. Na slici 4-12 možete vidjeti dvije takve situacije. U prvoj nastojimo identificirati ime kontrole i Visual Studio nam javlja da se radi o kontroli tipa System.Windows.Forms.Label te da se ona nalazi na formi Form1. U drugoj situaciji pokazivač miša držimo nad svojstvom Text i saznajemo koji tip vrijednosti mu valja pridružiti (string), na koji tip kontrole se odnosi (riječ Control znači da se odnosi na sve kontrole), te čak pruža kratak opis svojstva.

Slika 4-13: Ukazivanje na greške u sintaksi

Na slici 4-13 prikazan je jednostavan primjer kako Visual Studio ukazuje na greške u sintaksi. Crvenom valovitom crtom označeno je mjesto na kojem sasvim izvjesno nedostaje znak točka-zarez. Dakako, ovakva pomoć uskočit će i u nešto složenijim primjerima.



Slika 4-14: Pomoćni prozor Task List

Pozabavimo se na kraju malo i pomoćnim prozorom Task List. On je zamišljen kao mjesto gdje programer može zapisati stvari koje još mora napraviti, ali i gdje će kompajler zapisati greške i upozorenja na koje naiđe. Međutim, ovdje ga spominjemo zbog zanimljiva načina pomoću kojega možete dodavati zadatke u prozor. Dovoljno je napisati komentar u ovom formatu:

```
// TODO: Tekst koji želimo da se pojavi u pomoćnom prozoru
```

...i on će se pojaviti u pomoćnom prozoru. (Ukoliko se ne pojavi, desnom tipkom miša kliknite na listu zadataka i postavite opciju Show Tasks na vrijednost All.) Osim riječi TODO, u konfiguraciji programa (Tools > Options > Environment > Task List) možete dodati i druge ključne riječi tog tipa, tzv. tokene.

Već smo spomenuli da se automatski generiran kôd skriva iza pravokutnika s natpisom "Windows Form Designer generated code". Želite li i vi sakriti dio kôda na isti način, potrebne su vam ključne riječi "#region" i "#endregion". Evo primjera:

```
#region Neke funkcije
private void button1_Click(object sender, System.EventArgs e)
{
   // neki kôd
}
#endregion
```

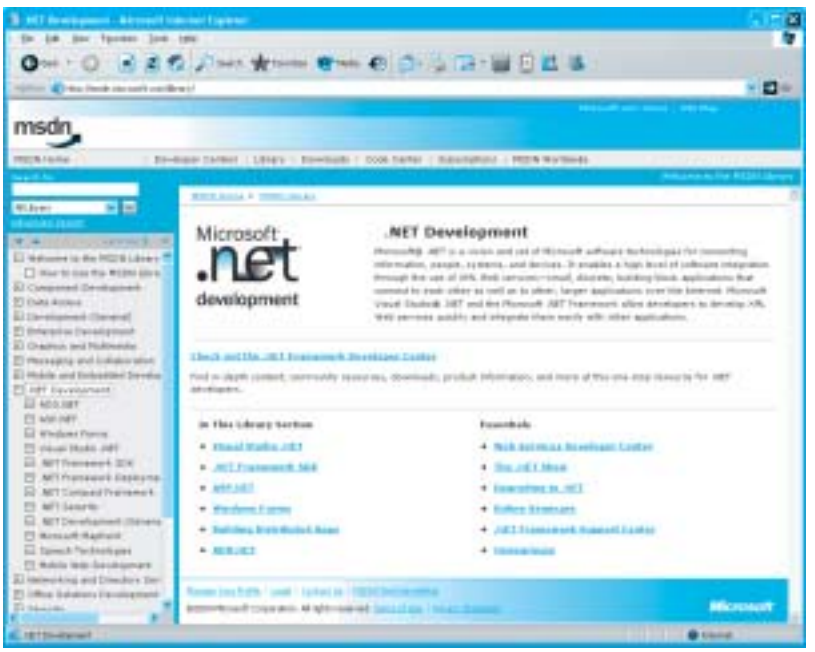
Njihovim dodavanjem ništa se ne mijenja na funkcionalnosti, jedino se lijevo od početka bloka, pokraj broja reda, pojavljuje kvadratić sa znakom minusa, pritiskom na koji skrivate označeni blok. Osim ručno označenih blokova, na isti je način moguće sakriti funkcije i druge automatski detektirane blokove koji imaju kvadratić s minusom u redu svog početka.

Visual Studio skriva još mnoge sitnice koje će vam pomoći u radu. Neke od njih, poput mogućnosti *Search and Replace* nismo niti spominjali jer vjerujemo da su vam poznate iz drugih programa, neke ćete upoznati u kasnijim poglavljima, a neke ćete morati otkriti sami.

Dokumentacija

Pažnja koju Microsoft posvećuje informatičkim profesionalcima koji u radu koriste njegove proizvode i tehnologije oduvijek je bila na visokoj razini. Kao jednu od najvažnijih komponenti te pažnje moramo izdvojiti Microsoft Developer Network (skraćeno MSDN), koji predstavlja set *online* i *offline* usluga namijenjenih programerima kako bi im pisanje aplikacija pomoću Microsoftovih proizvoda i tehnologija bilo što jednostavnije.

Dio MSDN-a koji se bavi dokumentacijom naziva se MSDN Library. Kao što smo već spomenuli, MSDN Library je dostupan na dva načina – na optičkom mediju (nekoliko CD-a ili jedan DVD) i na Internetu na adresi http://msdn.microsoft.com/library/.



Slika 4-15: Internetsko izdanje MSDN Libraryja

Tekstovi dostupni u MSDN Libraryju ne svode se samo na specifikacije već i na velik broj primjera, stručnih članaka, vodiča i tekstova u raznim drugim formama. Sve je tekstove moguće pretraživati, no kako se radi o izuzetno velikoj količini informacija, puno se bolji rezultati postižu navigacijom kroz dostupne kategorije.

Kategorija u kojoj ćete kao programer u .NET-u najviše prebivati jest ".NET Development" (vidi sliku 4-15), pa preporučujemo da je razgledate i steknete dojam kako je organizirana. Takva "ogledna šetnja" olakšat će vam kasnije snalaženje, kada budete rješenja rješenja za konkretne probleme.

Tablica 4-3: Putanje do nekih važnijih tema u MSDN Libraryju

Tema	Putanja do teme
Uvod u .NET Framework	.NET Development > .NET Framework SDK > .NET Framework
Referenca jezika C#	.NET Development > Visual Studio > Product Documentation > Visual Basic and Visual C# > Reference > Visual C# Language > C# Programmer's Reference
Bazna biblioteka klasa	.NET Development > .NET Framework SDK > .NET Framework > Reference > Class Library
Prozorske aplikacije	.NET Development > Windows Forms
Rad s bazama podataka (ADO.NET)	.NET Development > ADO.NET
Web-aplikacije (ASP.NET)	.NET Development > ASP.NET

Kad se udomaćite u MSDN Libraryju, zavirite i u ostatak stranica MSDN-a koje se nalaze na adresi http://msdn.microsoft.com/. Tamo ćete naći još hrpu tekstova, opisa, primjera, ideja, linkova i objašnjenja. Posebno valja izdvojiti dio nazvan Downloads, u kojem možete naći zanimljive i korisne dodatke za Visual Studio te veliku bazu primjera pod nazivom MSDN Code Center.

Osim u MSDN-u, dokumentaciju, primjere i pomoć možete tražiti na brojnim web-stranicama koje se bave tematikom vezanom uz .NET platformu. Ne valja podcijeniti niti potencijal grupa na Usenetu (nntp://microsoft.public.dotnet.*), a u praksi se posebno korisnim pokazalo pretraživanje arhiva tamo objavljenih članaka pomoću tražilice na adresi http://groups.google.com/.

Osim javnih Usenetskih grupa, postoje i "privatne", i to na hrvatskom jeziku. Riječ "privatne" u ovom slučaju ne znači da njima ne možete pristupati, nego da im se mora pristupati preko posebnog poslužitelja. Detaljne upute i parametre za spajanje potražite na adresi http://www.microsoft.com-/croatia/support/newsgroups/.

Slika 4-16:
Konfiguracija Outlook Expressa za praćenje privatnih hrvatskih Usenet grupa (parametri se povremeno mijenjaju, za aktualno važeće posjetite web-adresu u tekstu)

II dio

Osnove programiranja

5. POGLAVLJE

Tipovi podataka

U ovom poglavlju:

- Tipovi podataka u .NET-u
- Pretvaranje tipova podataka
- Što su implicitne, a što eksplicitne pretvorbe
- Korištenje konstanti i pobrojanih članova
- Rad s poljima i kolekcijama
- Što su klase i strukture
- Korištenje delegata

Temelj su svakog programa, ma kako jednostavan bio, podaci s kojima radi. To mogu biti obični brojevi u programima koji obavljaju jednostavne kalkulacije, znakovni nizovi u programima koji komuniciraju s korisnikom ili pak neke složenije strukture, poput specifičnih objekata koji predstavljaju nekakav entitet ili polja, koja predstavljaju povezani niz podataka.

Da biste mogli u svojim programima koristiti različite vrste podataka, trebate upoznati tipove podataka koje vam se nude u .NET-u. Imate li pak nekog programerskog iskustva, brzo ćete se snaći i uočiti sve novitete i različitosti, a sve će vam to itekako trebati u programiranju aplikacija za .NET.

Korištenje tipova podataka

U .NET Frameworku postoji vrlo precizno definiran sustav tipova podataka. Svi .NET jezici su *strongly typed* – znači da su svi podaci čvrsto povezani sa svojim tipom i da im nije moguće slobodno mijenjati tip i razmjenjivati ih između objekata različitih tipova.

Naravno, zato postoji mogućnost konverzija podataka, koja se može odvijati implicitno (što znači da je nije potrebno navoditi, već se podrazumijeva) ili eksplicitno (tako da vi pozovete metodu za konverziju podataka). Tipičan primjer događa se pri spajanju brojeva i tekstova – želite li ih sve spojiti u jedan veći tekst, trebat ćete prvo eksplicitno pretvoriti sve brojeve u tekstualni tip podataka.

Tipovi podataka

Dakle, tipovi podataka određuju vrstu podataka koju koristite i pohranjujete u svom programu. Tipovi podataka se prvenstveno dijele u reference i vrijednosti – reference su podaci kojima nije uvijek poznata veličina te zato varijable reference ukazuju na neku lokaciju u memoriji na kojoj je spremljen podatak, dok *vrijednosti* uvijek zauzimaju istu količinu memorije (ovisno o svom tipu, primjerice svaka vrijednost *int* zauzima 4 bajta) te njihove varijable direktno sadrže njihovu vrijednost.

Primjerice, znakovne nizove uvijek možete povećavati, čime dakako zauzimate veću količinu memorije, te su zato oni reference – varijable ukazuju na njihovu lokaciju u memoriji. S druge strane, brojevi uvijek zauzimaju određenu količinu memorije (koja ovisi o tipu broja, odnosno njegovu rasponu, pa mogu zauzimati od 1 do 8 *byteova*) te je moguće već prije rezervirati točnu količinu memorije za njih. Takve varijable uvijek sadrže vrijednost, a ne pokazivač na mjesto u memoriji otkud počinje neodređeno velik podatak (kao što je slučaj sa znakovnim tipovima).

Vrijednosni podaci se najjednostavnije mogu podijeliti na cjelobrojne, decimalne i logičke tipove te tipove čiji se podaci sastoje od samo jednog znaka (odn. podaci tipa *char*). Reference se dijele na znakovne nizove (tip *string)* i objekte (ili tip *Object*).

Cjelobrojni tipovi

U tablici 5-1 nalazi se popis dostupnih cjelobrojnih tipova podataka i njihovih raspona.

Vjerojatno ste u tablici 5-1 uočili nepoznate pojmove – *signed* i *unsigned*. Ukoliko je neki broj *signed*, to znači da se jedan njegov bit koristi za predznak te stoga može poprimiti i pozitivne i negativne vrijednosti. Za razliku od njega, *unsigned* tip ne koristi jedan bit za predznak te stoga može poprimiti samo pozitivne vrijednosti, ali zato doseže višu najveću pozitivnu vrijednost.

Tablica 5-1:
Popis cjelobrojnih tipova podataka i njihovih raspona

Tip	C# ime	Opis	Raspon
System.Byte	byte	8-bitna unsigned vrijednost (1 bajt)	0 do 255
System.Int16	short	16-bitna signed vrijednost (2 bajta)	-32768 do 32767
System.Int32	int	32-bitna signed vrijednost (4 bajta)	-2^{31} do $2^{31}-1$
System.Int64	long	64-bitna signed vrijednost (8 bajtova)	-2^{63} do $2^{63}-1$
System.Sbyte	sbyte	8-bitna signed vrijednost (1 bajt)	-128 do 127
System.UInt16	ushort	16-bitna unsigned vrijednost (2 bajta)	0 do 65535
System.UInt32	uint	32-bitna unsigned vrijednost (4 bajta)	0 do $2^{32}-1$
System.UInt64	ulong	64-bitna unsigned vrijednost (8 bajtova)	0 do $2^{64}-1$

Kako se cijela struktura .NET organizira u hijerarhiju klasa, tako i svaki podatak ima odgovarajuću klasu, koja je u tablici 5-1 navedena u stupcu *Tip*. Dakle, najočitije bi bilo na sljedeći način deklarirati varijable:

```
System.Int32 broj = new System.Int32();
broj = 15;
```

No to je ipak malčice prekomplicirano za korištenje, pogotovo pri radu s nečim tako jednostavnim kao što su varijable. Zato u C#-u postoji odgovarajuća ključna riječ koja služi za deklaraciju varijabli, a navedena je u stupcu *C# ime*. Koristeći je možete mnogo lakše deklarirati varijable:

```
int broj;
broj = 15;
```

Ako baš želite, cjelobrojnim tipovima podataka možete pridjeljivati i heksadecimalne vrijednosti tako da im dodate prefiks 0x. Primjerice, želite li kao u prethodnom primjeru postaviti varijablu broj na 15, napisat ćete:

```
broj = 0xF;
```

Tipovi s pomičnim zarezom

Želite li u svojim programima koristiti realne brojeve (ili decimalne, odnosno brojeve s pomičnim zarezom), koristit ćete neki od tipova iz tablice 5-2.

Tablica 5-2:
Popis tipova s pomičnim zarezom te njihove preciznosti i rasponi

Tip	C# ime	Opis	Preciznost	Približni raspon
System.Single	float	32-bitna vrijednost	7 značajnih znamenki	$-3.4*10^{38}$ do $-1.4*10^{-45}$ i $1.4*10^{-45}$ do $3.4*10^{38}$
System.Double	double	64-bitna vrijednost	15-16 značajnih znamenki	$-1.7*10^{308}$ do $-5.0*10^{-324}$ i $5.0*10^{-324}$ do $1.7*10^{308}$
System.Decimal	decimal	128-bitna vrijednost	28 značajnih znamenki	$-7.9*10^{28}$ do $-1.0*10^{-28}$ i $1.0*10^{-28}$ do $7.9*10^{28}$

Tip podataka *System.Single* prikladan je za jednostavne izračune koji ne zahtijevaju veliku preciznost rezultata, no ukoliko vam je ona relativno bitna, upotrijebit ćete tip *System.Double* koji, uz povećanu preciznost, može spremiti brojeve ogromnog raspona. Bavite li se izrazito preciznim izračunima, morat ćete ipak upotrijebiti vrijednost *System.Decimal,* koja pruža daleko najveću preciznost rezultata (čak 28 značajnih znamenki)

Logički tip

Želite u svojem programu koristiti jednostavni tip podataka, koji omogućava spremanje samo vrijednosti *istinito* ili *neistinito*, koristit ćete *System.Boolean* odnosno tip podataka *bool*.

```
bool logika;  // deklaracija varijable "logika"
logika = true; // postavljanje na istinitu vrijednost

if (logika)   // ili "if (logika == true)"
{
  Console.WriteLine("Istinito!");
}
```

Preciznost ili nepreciznost, pitanje je sad

U tablici 5-2 nalazi se pojam preciznosti vrijednosti odnosno broj značajnih znamenki (engl. *significant digit*). Objasnimo sve na broju 4215.02474. Ukoliko se broj spremi s preciznošću od dvije značajne znamenke, dobit ćemo 4200; ukoliko se koristi preciznost od tri značajne znamenke, dobit ćemo 4220; ukoliko se pak koristi 5 značajnih znamenki, dobit će se 4215.0; ukoliko se koristi 7 značajnih znamenki, dobivamo 4215.025.

Ukoliko takve vrijednosti prikažete s jednom znamenkom prije točke (kako se i spremaju u računalu), odnosno $4.2*10^3$, $4.22*10^3$, $4.2150*10^3$ i $4.215025 * 10^3$, uočit ćete primjenu i svrhu značajnih znamenki.

Dakle, dok *float* podaci mogu pamtiti brojeve s preciznošću od 7 znamenki (primjerice, $4.215025*10^3$), *decimal* je mnogo točniji i može spremiti broj poput $4.21502474920239135819235 7189*10^3$ (naravno, uočite da 10^3 uopće nije bitno za primjer i da tu može stajati bilo koja potencija, sve dok je konačni broj unutar odgovarajućeg raspona).

Naravno, osim istinite vrijednosti ili *true*, u C#-u možete definirati i neistinitu vrijednost odnosno *false*. Logički tip podataka je posebno koristan u *if*-naredbama i sličnim provjeravanjima ispunjenosti određenih uvjeta.

Znakovni tipovi

U .NET-u imate mogućnost raditi s dva znakovna tipa – dok jedan služi za rad samo s jednim znakom (*char* tip), drugi koristi za spremanje znakovnih nizova (tip *string*). Vrijednosti varijabli tipa *char* upisujete korištenjem jednostrukih navodnika (‘), a podatke tipa *string* upisujete korištenjem dvostrukih navodnika.

```
char mojZnak;
mojZnak = 'A';

string mojNiz;
mojNiz = "Dobar dan!";
```

Kako su znakovni nizovi podatak koji ćete najviše koristiti u komunikaciji s korisnikom za ispis podataka, nužno je poznavati njegove mogućnosti. Primjerice, spajanje dvaju nizova u jedan zove se konkatenacija i obavlja se pomoću “+” operatora, baš kao i pri zbrajanju brojeva.

```
string mojNiz1 = "Dobar ";
string mojNiz2 = "dan!";
string mojNiz3;

mojNiz3 = mojNiz1 + mojNiz2; // "Dobar dan!"
```

Naravno, tip podataka *string* ima ugrađen cijeli niz metoda koje možete koristiti pri radu, a važnije su opisane u tablici 5-3.

Tablica 5-3: ***Važnije metode string tipa podataka***

Metoda	Opis
Insert	umeće neki znakovni niz na određenu poziciju trenutnog znakovnog niza
PadLeft, PadRight	dodaju znakove na lijevu stranu (početak) i desnu stranu (kraj) niza
Remove	briše određen broj znakova iz niza
Replace	zamjenjuje sva pojavljivanja nekog znaka u nizu s nekim drugim znakom
Substring	vraća podniz znakovnog niza
ToLower, ToUpper	pretvaraju sve znakove niza u mala ili velika slova
Trim	briše sve suvišne *whitespace* znakove (razmaci, tabovi, itd.) s početka i kraja niza

Sve opisane metode koriste se nad nekom varijablom tipa *string*, kao što je prikazano na slici 5-1. Primjerice, da biste ispisali prvih pet znakova nekog niza (zbog jednostavnosti, nastavit ćemo prethodni primjer), iskoristit ćete metodu *Substring*:

```
string mojNiz = mojNiz3.Substring(0, 5);
```

Metoda *Substring* prima dva parametra – prvi je znak od kojeg počinje podniz (0 označava prvi znak, 1 je drugi itd.), a drugi je broj znakova podniza. Stoga, primijenimo li prethodnu naredbu nad nizom "Dobar dan!", rezultat će biti "Dobar".

U svojim znakovnim nizovima možete koristiti i različite posebne znakove (tzv. *escape-znakove*). Primjerice, želite li u aplikaciji u konzoli izbrisati jedan znak s ekrana, ispisat ćete znak *backspace* koji će učiniti baš to. Želite li prijeći u novi red, ispisat ćete tzv. znak *newline*. Neki od tih znakova opisani su u tablici 5-4.

Znakovni niz	Opis
\'	jednostruki navodnik
\"	dvostruki navodnik
\\	znak backslash
\b	znak backspace za brisanje znaka lijevo od pokazivača
\n	znak newline za prelazak u naredni redak
\t	znak za pomak od nekoliko mjesta (znak tab)

Tablica 5-4: ***Neki od posebnih znakova koje možete koristiti u svojim znakovnim nizovima***

Uočite da se u tablici 5-4 koristi znak *backslash* za označavanje svih posebnih znakovnih nizova (naprimjer: \", \t itd.). Stoga je potreban i poseban znakovni niz ukoliko se želi ispisati sam *backslash* – \\.

Primjerice, da biste unutar niza ispisali jednostruke i dvostruke navodnike, morat ćete iskoristiti sljedeću sintaksu:

```
// Uš'o medo u dućan i reče: "Dobar dan!"
mojNiz = "Uš\'o medo u dućan i reče: \"Dobar dan!\"";
```

Objekt

Tip podataka *System.Object* je glavni tip podataka u .NET Frameworku i svi drugi tipovi su izvedeni iz njega. On ima 4 metode, koje zatim nasljeđuju svi drugi tipovi podataka i mogu se koristiti nad bilo kojom varijablom u .NET-u: *Equals* (provjerava jednakost između dvije instance), *GetHashCode* (vraća *hash* kôd), *GetType* (vraća objekt *type* koji govori o kojem se tipu podataka radi) i *ToString* (vraća tekstualni opis objekta).

Korištenje tipa *object* jednostavno je i u njega se mogu spremiti bilo kakve vrijednosti:

```
object mojObjekt;
mojObjekt = "Dobar dan!";
mojObjekt = 500;
mojObjekt = new System.Xml.XmlDocument();
```

No da biste mogli koristiti funkcionalnost pojedinog tipa podataka, morat ćete objekt pretvoriti u taj tip. Primjerice, da biste mogli u prvom primjeru raditi s podnizovima ili zamjenjivati znakove,

morat ćete ga pretvoriti u objekt *string*. Slično ćete i drugi primjer morati pretvoriti u neki brojčani tip da biste ga mogli koristiti u računskim operacijama. Naravno, isto vrijedi i za treći primjer, koji biste trebali pretvoriti u tip *XmlDocument*.

Ključna stvar kod korištenja tipa podataka *object* je *boxing*. Radi se, naime, o implicitnoj konverziji vrijednosnih tipova podataka u reference. Primjerice, kako tip *object* predstavlja nadtip svim podacima u .NET-u, sve varijable možete pretvoriti u tip *object* i koristiti ih kao reference. Evo i primjera:

```
int Broj = 100;
object Objekt;
Objekt = Broj;
```

Dakle, cjelobrojnu varijablu smo pretvorili u objekt tako što smo njenu vrijednost pridružili objektu. Želimo li pak provjeriti što se nalazi u samoj *kutiji*, tj. objektu, možemo napisati:

```
if (Objekt is int)
{
  Console.WriteLine("Objekt sadrži broj!");
}
```

Suprotna operacija odnosno *unboxing* je pretvaranje natrag u originalni tip podataka. To je pak eksplicitna operacija odnosno morate u kôdu naznačiti da želite provesti pretvorbu podataka, za razliku od operacije *boxing* koja je implicitna, jer samo trebate pridijeliti varijablu objektu i automatski će se dogoditi pretvorba.

Pretvaranje tipova podataka

Mnogo puta ćete u svom kódu koristiti podatke koje je potrebno pretvoriti u neki drugi tip. Recimo, radit ćete s brojevima, obavljati različite operacije, a zatim ćete htjeti ispisati poruku "Konačni rezultat je xyz". Očito ćete dobiveni broj morati pretvoriti u znakovni niz i zatim ga spojiti s tekstom "Konačni rezultat je".

To je samo jedna od primjena. Ponekad ćete obavljati i pretvorbe među sličnim tipovima podataka, koje nisu previše uočljive, a imaju velik utjecaj na kôd. Tako ćete ponekad pretvarati cijele brojeve u one s pomičnim zarezom da biste dobili točan rezultat pri, recimo, dijeljenju.

Konverzija podataka iz jednog tipa u drugi može biti obavljena na dva načina – *implicitno*, što znači da se konverzija obavlja automatski, i *eksplicitno*, što znači da se konverzija obavlja na vaš zahtjev odnosno korištenjem posebnih naredbi u kódu.

Implicitne pretvorbe

Implicitna pretvaranja između tipova podataka obavljaju se uvijek kad pri pretvaranju ne može doći do gubitka podataka.

```
int Broj1 = 50;
long Broj2;

Broj2 = Broj1;
```

Kako tip brojeva *long* ima mnogo veći raspon nego običan tip *int*, pri konverziji neće sigurno doći do gubitka podataka. Tad se konverzija obavlja na jednostavan način – na mjestu gdje je bio očekivan tip podatka *long* (u primjeru je to iza znaka jednakosti, no može biti i na mjestu parametra pri pozivu metode itd.), postavi se tip podatka *int*, a pretvorba iz *int* u *long* se obavlja automatski.

U tablici 5-5 je popis mogućih implicitnih konverzija.

Tablica 5-5:
Popis konverzija koje će se obaviti implicitno

Izvorni tip podataka	Implicitna konverzija moguća u tipove
byte	short, ushort, int, uint, long, ulong, float, double, decimal
short	int, long, float, double, decimal
int	long, float, double, decimal
long	float, double, decimal
float	double
char	int, uint, long, ulong, float, double, decimal
sbyte	short, int, long, float, double, decimal
ushort	int, uint, long, ulong, float, double, decimal
uint	long, ulong, float, double, decimal ulong float, double, decimal

Svako pridjeljivanje neke varijable tipa podatka iz lijevog stupca varijabli koja je tipa navedenog u desnom stupcu obavit će se implicitno – bez ikakvog dodatno upozoravanja i bez gubitka podataka.

Eksplicitne pretvorbe

Ukoliko se neke pretvorbe ne mogu obaviti implicitno, trebat će ih obaviti korištenjem posebnih naredbi. Najjednostavnije je to napraviti *castanjem* neke varijable u drugi tip podataka. To se obavlja tako da se ispred same varijable u zagradama stavi novi tip podataka.

```
long Broj1 = 50;
int Broj2;

Broj2 = (int) Broj1;
```

Primijetite da dok je pretvaranje iz *int* u *long* moguće, jer pritom nema gubitaka podataka, prethodni primjer je riskantan – iako i *long* i *int* mogu spremiti vrijednost 50 korištenu u primjeru, da se koristila neka vrijednost izvan opsega tipa *int* došlo bi do gubitka podataka. Pogledajte primjer:

```
int Broj1 = 10000;
byte Broj2;

Broj2 = (byte) Broj1;
```

Kako je opseg tipa podatka *byte* mnogo manji od vrijednosti koja se u njega pokušava spremiti (10000), doći će do gubitka podataka – ispišete li varijablu *Broj2*, uvidjet ćete da je u njoj spremljena vrijednost 16.

Kao što ste vidjeli u primjerima, lako se dogodi gubitak podataka pri eksplicitnim konverzijama. Budite vrlo pažljivi pri takvim pretvorbama i koristite ih samo kad ste sigurni u vrijednosti koje će biti spremljene. Također, preporučuje se i ugradnja mehanizma za hvatanje iznimaka, koje se mogu dogoditi ukoliko se pokuša izvršiti nedozvoljena eksplicitna konverzija. O tim ćete mehanizmima naučiti više u sljedećim poglavljima.

Ponekad ćete poželjeti obaviti pretvorbe za koje ne možete utvrditi hoće li pri njima doći do gubitka podataka, jer se to može znati samo za slične tipove. No što ako želite pretvoriti broj u znakovni niz ili obrnuto? Tad je nemoguće znati hoće li doći do gubitka i nužno je koristiti posebnu naredbu za konverziju.

Za to će vam poslužiti klasa *Convert* i njene metode. Njome vam je omogućeno pretvaranje između svih tipova:

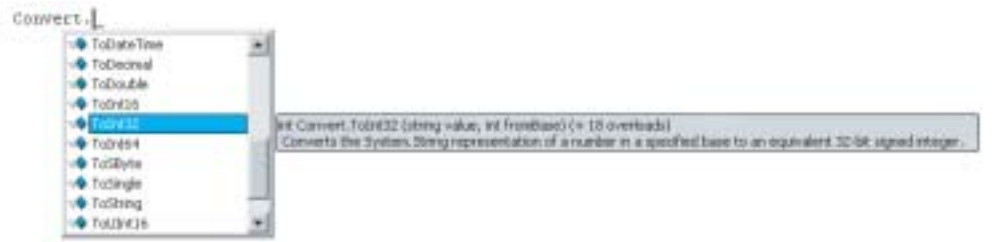
```
int Broj = 100;
string Niz;

Niz = Convert.ToString(Broj);
```

Dakle, treba samo pozvati odgovarajuću metodu za pretvaranje i za parametar joj proslijediti varijablu koju želite pretvoriti.

Konstante, polja i kolekcije

Dosad ste upoznali sve osnovne tipove podataka u .NET-u, no često će vaše aplikacije zahtijevati i složenije tipove, bilo zbog jednostavnosti i lakšeg korištenja ili zbog elegancije u programiranju.

Slika 5-2:
Pregršt metoda za pretvaranje u klasi Convert



Za pretvaranje u znakovni niz mnogo je jednostavnije iskoristiti metodu ToString() koja je dostupna svakoj varijabli, primjerice:

```
int Broj = 100;
string Niz = Broj.ToString();
```

Nju možete koristiti i pri konkateniranju nekog broja sa znakovnim nizom. Da biste ih spojili u jedan znakovni niz, morat ćete broj pretvoriti u tip *string*. To ćete često raditi pri ispisivanju, primjerice:

```
Console.WriteLine("Broj je jednak " + Broj.ToString());
```

Na narednim stranicama upoznat ćete se s poljima i kolekcijama, dvjema sličnim strukturama čija je glavna namjena grupiranje srodnih podataka. Krenimo ipak za početak s jednostavnijim stvarima – s konstantama.

Konstante

Na konstante se može gledati kao na varijable koje svoju vrijednost nikad ne mijenjaju. Njihova je primjena jasna – imate li vrijednosti koje su fiksne (primjerice, za računanje površine kruga, koristit ćete konstantnu varijablu u kojoj je spremljena vrijednosti PI), pridijelit ćete ih nekoj konstanti koju lako možete koristiti iz bilo kojeg mjesta u kódu. Želite li negdje kasnije u razvoju programa promijeniti tu konstantu, nećete trebati prolaziti cijeli kôd i mijenjati njenu vrijednost, već ćete samo promijeniti deklaraciju konstante i pridijeliti joj novu vrijednost.

Konstante se definiraju s ključnom riječi *const*. Njihovim korištenjem mnogo ćete se lakše snalaziti u kódu i lakše će vam biti pronaći greške.

```
const double Pi = 3.14159265;
const int BrzinaSvjetlosti = 300000;
const string ImePrograma = "Moj program";
```

Kao što uočavate, konstante su obične varijable ispred kojih je napisano *const*.

Konstantama ne možete mijenjati vrijednost u svom programu – pokušate li izvesti sljedeće naredbe, kompajler će vam javiti grešku:

```
BrzinaSvjetlosti = 299999;
ImePrograma = "Najbolji program";
```

Konstante možete koristiti u svom programu kao i sve druge varijable, uz spomenuto ograničenje koje vam zabranjuje mijenjanje njihovih vrijednosti.

Ukoliko stvarno želite koristiti konstantu PI, ne trebate pisati svoju verziju već možete iskoristiti predefiniranu u klasi *Math*, kojoj možete pristupiti s *Math.PI*.

Pobrojani članovi

Za pisanje čitkog i urednog kôda, osim korištenja konstanti, mogu vam pomoći i pobrojani članovi (engl. *enumerations*). Njihova je svrha vrlo slična konstantama – oni zapravo predstavljaju niz povezanih konstanti s logičnim imenima. Tipičan primjer su dani u tjednu – želite li u svom kódu svakom danu pridijeliti redni broj, možete koristiti 7 različitih konstanti, a možete ih povezati u pobrojen niz.

Za definiranje pobrojanog niza ključna je riječ *enum,* koja zapravo predstavlja tip podatka. Unutar vitičastih zagrada navode se elementi niza s odgovarajućim vrijednostima.

```
enum Tjedan
{
  Ponedjeljak = 1,
  Utorak = 2,
  Srijeda = 3,
  Cetvrtak = 4,
```

```
    Petak = 5,
    Subota = 6,
    Nedjelja = 7
}
```

Kao što primjećujete, radi se o konstantama koje se nalaze unutar pobrojanog niza *Tjedan*, a svaka ima svoju vrijednost.

Ukoliko ne navedete drugačije, sve će vrijednosti pobrojanog niza biti *int*. Naravno, vi imate na raspolaganju i definiranje drugačijeg tipa, pa pobrojani nizovi mogu biti i tipa *byte*, *short* ili *long*. Da biste definirali niz drugačijeg tipa, primjerice *short*, iskoristit ćete sljedeću sintaksu:

```
enum Tjedan : short
{
  // elementi niza
}
```

Imajte na umu da su pobrojani nizovi posebni tipovi podataka i da njih ne možete definirati unutar neke metode. Oni mogu biti definirani isključivo unutar neke klase, dakle izvan svih njenih metoda.

Naravno, sve pobrojane nizove možete jednostavno koristiti u svom kódu – umjesto da pišete vrijednost svake konstante, jednostavno ćete napisati ime pobrojenog člana. No budite oprezni – u C#-u ih morate prije korištenja obavezno pretvoriti u tip podataka koji sadržavaju.

```
int DanasnjiDan = 3;
// int DanasnjiDan = (int) Tjedan.Srijeda;

if (DanasnjiDan == (int) Tjedan.Srijeda)
{
  Console.WriteLine("Danas je srijeda!");
}
```

Vrlo je važno uočiti *castanje* u tip podatka *int* prije same usporedbe, jer takve podatke sadržava pobrojani niz. U drugoj liniji u komentarima je navedena naredba koja je po funkcionalnosti identična prvoj, no mnogo elegantnija i čitljivija.

Pobrojani nizovi su vrlo korisni i za stvaranje varijabli. Tako možete definirati varijablu koja je tipa *Tjedan*, a ona će moći sadržavati samo vrijednosti koje su definirane u pobrojanom nizu. Deklaracija takve varijable ni po čemu se ne razlikuje od drugih varijabli:

Slika 5-3: ***Visual Studio .NET će vam pri radu s pobrojanim nizovima ponuditi popis svih njihovih članova složenih po abecedi.***

```
Tjedan mojDanTjedna;
```

Takvim varijablama nećete moći pridijeliti vrijednost direktno pišući broj koji predstavlja vrijednost nekog dana, već ćete morati iskoristiti sam pobrojani niz.

```
mojDanTjedna = Tjedan.Petak;

if (mojDanTjedna == Tjedan.Utorak)
{
  // itd.
}
```

Već vjerojatno i sami uočavati koliko je čitljiviji ovakav kôd, za razliku od onog u kojem biste direktno koristili vrijednosti. Imate li u svom kódu potrebu za ograničavanjem neke varijable samo na određen set mogućih vrijednosti (u našem primjeru, varijabla *mojDanTjedna* mogla bi poprimiti samo neku od vrijednosti iz pobrojanog niza *Tjedan*), pobrojani nizovi su jedini pravi izbor.

Kao što ćete vidjeti u narednim poglavljima, u baznim klasama .NET-a postoji mnoštvo pobrojanih nizova koji uglavnom spremaju vrijednosti o mogućim stanjima ili svojstvima nekog objekta, stoga je bitno znati o čemu se radi i kako sami možete načiniti pobrojane nizove.

Ponekad nije potrebno eksplicitno ispisati vrijednosti svakog člana pobrojanog niza. Ukoliko ne navedete drukčije, prvi član će imati vrijednost 0, drugi će imati vrijednost 1, treći 2 itd.

```
enum Brojevi
{
  Nula,  // 0
  Jedan, // 1
  Dva    // 2
}
```

Polja

Za razliku od pobrojanih nizova u kojima su sve vrijednosti bile predefinirane, u programiranju možete koristiti i polja (u literaturi još možete pronaći i izraz *matrice*, od engl. *array*). Radi se o grupi srodnih vrijednosti istog tipa, kojima možete pristupati korištenjem njihovog indeksa odnosno rednog broja.

Uzmimo jednostavan primjer: ukoliko želite u svojoj aplikaciji pobrojati sve učenike nekog razreda, iskoristit ćete polje. Definirat ćete da broj članova polja odgovara broju učenika u razredu, a zatim ćete za svaku vrijednost polja upisati ime pojedinog učenika u razredu. Tada ćete sve učenike razreda imati u samo jednoj varijabli, u polju, a svakom učeniku ćete moći pristupati prema njegovom rednom broju u polju.

Upamtite – za polja u C#-u (kao i u većini drugih programskih jezika) kažemo da su *zero-based*. To znači da prvi član polja ima indeks 0, drugom se može pristupiti korištenjem indeksa 1 itd.

Deklaracija i inicijalizacija polja razlikuje se od deklaracije običnih varijabli. Tako ćete morati navesti broj članova polja unutar uglatih zagrada, a uglate zagrade ćete morati i staviti kod tipa polja. Primijetite da svako polje ima svoj tip koji određuje njegove članove. Tako će polje *int* moći sadržavati samo cjelobrojne vrijednosti, polje *string* samo znakovne nizove itd.

Nemojte da vas zbuni broj članova koji navodite pri deklaraciji polja. On označava ukupan broj članova, a zbog činjenice da prvi član polja ima indeks 0, zadnji će član imati indeks za jedan manji od ukupnog broja članova. Sljedeći primjer definira polje koje ima 10 članova, s indeksima od 0 do 9.

```
int[] mojePolje = new int[10];
```

Prethodni je primjer tako identičan dužoj verziji po kojoj deklarirate polje u jednoj liniji, a inicijalizirate na određenu veličinu u narednoj liniji (naravno, te linije mogu biti međusobno udaljene u kódu):

```
int[] mojePolje;

// ...

mojePolje = new int[10];
```

Veličinu nekog polja možete bilo kad u kódu promijeniti, no pripazite jer u tom slučaju gubite vrijednosti svih članova polja. Sintaksa promjene veličine polja jednaka je njegovoj početnoj inicijalizaciji:

```
mojePolje = new int[20];
```

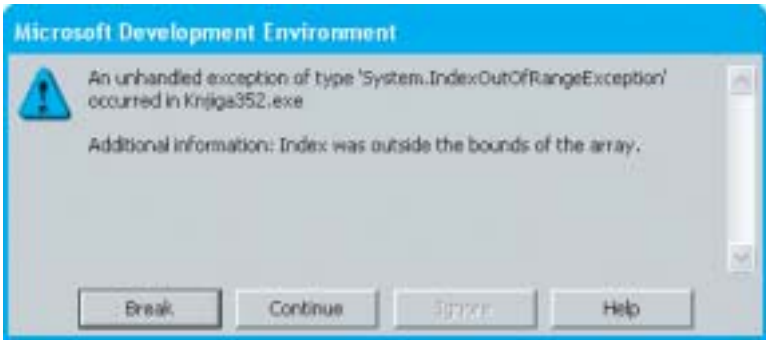
Vrijednosti polja pišete i čitate korištenjem njihovih indeksa.

```
mojePolje[0] = 50;
mojePolje[1] = 34;
mojePolje[2] = 17;

// Ispisat će "34"
Console.WriteLine(mojePolje[1]);

mojePolje[10] = 12; // Greška!
```

Ukoliko pokušate pristupiti nepostojećem članu polja korištenjem indeksa koji je izvan dopuštenih granica (u prethodnom primjeru indeks 10 je nevaljan – iako postoji 10 članova polja, njihovi se indeksi kreću od 0 do 9), kompajler vam neće javiti grešku, već će se ona dogoditi pri izvršavanju programa, stoga treba biti veoma oprezan!

Želite li pak ispisati veličinu polja, na raspolaganju vam stoji svojstvo *Length* varijable polja. Ono je tipa *int*, jer sadrži broj, pa ćete ga morati pretvoriti u tip *string,* ukoliko ga želite ispisati uz poruku:

```
Console.WriteLine(mojePolje.Length);
Console.WriteLine("Polje ima " + mojePolje.Length.ToString() + " članova.");
```



Slika 5-4: ***Greška koja se pojavljuje pri izvršavanju programa kad se pokuša pristupiti nepostojećem članu polja***

Svojstvo *Length* možete koristiti i za ispis svih članova polja. Jednostavno ćete proći od nultog člana do zadnjeg člana (kojem je indeks za jedan manji od broja članova polja) i ispisati članove polja korištenjem brojača za označavanje indeksa:

```
for (int i = 0; i <= mojePolje.Length – 1; i++)
{
  Console.WriteLine(mojePolje[i]);
}
```

Priča o podjeli tipova podataka na vrijednosne tipove koji sadržavaju vrijednost podataka (npr. *int, bool, long*) i reference koje sadrže samo pokazivač na vrijednost podatka smještenu negdje drugdje u memoriji (*object* i *string*) sad postaje važna. Ukoliko deklarirate da polje sadržava 10 podataka vrijednosnog tipa, u memoriji će se zauzeti 10 mjesta i oni će automatski biti postavljeni na svoju početnu vrijednost – primjerice, ukoliko definirate polje od 10 vrijednosti tipa *int*, zapravo ćete stvoriti polje koje za 10 članova ima vrijednost 0. No ukoliko definirate polje referenci (npr. *string*), podaci neće imati postavljenu početnu vrijednost. To je i očekivano, jer će svaki član polja zapravo biti referenca na neku lokaciju u memoriji koja sadržava samu vrijednost. Kako vi niste eksplicitno postavili vrijednost svakog člana, radit će se o *null*-pokazivačima, koji, kao što im i ime kaže, ne pokazuju nikamo.

Prethodni su primjeri radili samo s jednodimenzionalnim poljima, no vi na to niste ograničeni. U .NET-u imate podršku za pravokutna i zupčasta polja, a nazvana su po svom obliku. Radi se o višedimenzionalnim poljima – sve ćemo objasniti na lako shvatljivom primjeru od dvije dimenzije, no sve rečeno vrijedi i za više dimenzija.

Pravokutna polja

Na dvodimenzionalno polje se može gledati kao na tablicu koja se sastoji od redaka i stupaca. U pravokutnom polju svi će reci imati jednak broj stupaca, što i očekujemo od tablice. Evo kako biste definirali polje od 2 retka i 4 stupca:

```
int[ , ] mojaTablica = new int[2, 4];
```

Uočite da sad unutar uglatih zagrada postoje dva parametra – pri definiranju tipa oni su ostavljeni praznima, a pri inicijaliziciji su u njih upisane dimenzije. Članovima polja pristupate na očekivani način:

```
mojaTablica[0,0] = 0;
mojaTablica[1,3] = 5;
```

Umjesto upisivanja dimenzija pri inicijalizaciji polja, možete odmah upisati same vrijednosti polja, čime direktno dajete do znanja dimenzije. Primjerice, jednodimenzionalno polje od 4 člana možete definirati na jedan od sljedeća dva načina:

```
int[] mojePolje = new int[] {1, 2, 3, 4};
int[] mojePolje = {1, 2, 3, 4}
```

Dakle, za definiranje članova polja pri inicijalizaciji koriste se vitičaste zagrade. Na sličan način možete definirati i dvodimenzionalno polje 3 x 3 (indeksi, dakle, idu od 0 do 2, a vrijednosti će biti izračunate kao tablica množenja):

```
int[ , ] mojaTablica = { {1, 2, 3}, {2, 4, 6}, {3, 6, 9} };
```

Kad smo spomenuli da možete definirati višedimenzionalna polja, nismo se šalili – naime, sljedeći primjer definira polje od pet dimenzija, 5 x 3 x 8 x 1 x 9, no upitno je kako biste se u njemu snašli zbog teškog predočivanja više dimenzija.

```
int[ , , , , ] petDimenzija = new int[5, 3, 8, 1, 9];
```

Zupčasta polja

Zupčasta polja mnogo su slobodnija što se tiče dimenzija. Zupčasto polje od dvije dimenzije nema isti broj elemenata u recima. Primjerice, da bi učenik prikazao svoje ocjene iz nekoliko predmeta, morat će koristiti zupčasto polje, jer neće svaki predmet imati isti broj ocjena. Prvi predmet (redak) će imati, recimo, 3 ocjene (stupca), drugi predmet će imati 6 ocjena, a treći samo 1 ocjenu.

Zbog takve specifičnosti, stvaranje zupčastih polja razlikuje se od stvaranja običnog polja. Na to se može gledati kao na stvaranje jednodimenzionalnog polja koje za svoje članove ima jednodimenzionalna polja. Tako će prvi redak sadržavati polje od 3 elementa, drugi polje od 6 elemenata, a treći polje od 1 elementa.

Evo kako biste definirali takvo polje:

```
int[][] mojeOcjene = new int[3][];
mojeOcjene[0] = new int[] { 5, 4, 5 };
mojeOcjene[1] = new int[] { 3, 4, 4, 5, 3, 5 };
mojeOcjene[2] = new int[] { 4 };
```

Želite li saznati neku od dimenzija, jednostavno iskoristite *Length* svojstvo:

```
// Ispisat će "3"
Console.WriteLine(mojeOcjene.Length);

// Ispisat će "6"
Console.WriteLine(mojeOcjene[1].Length);
```

Kolekcije

Ukoliko ste počeli raditi s poljima, vrlo brzo ste naišli na njihov nedostatak. Naime, poljima niste mogli mijenjati veličinu tako da ostanu sačuvane sve vrijednosti koje sadržavaju (primjerice, povećati broj članova polja s 10 na 20 da ostanu zapisane vrijednosti prvih 10 članova). To je zapravo vrlo važno – često na početku izvršavanja ne znate koliko će članova imati neko polje, već želite dodavati članove po potrebi.

Kolekcija	Opis
BitArray	kolekcija bitova (0 ili 1)
Hashtable	kolekcija parova *ključ* i *vrijednost* organiziranih po *hash*-kódu ključa
Queue	red; kolekcija organizirana na FIFO principu (*first-in*, *first-out*)
SortedList	kolekcija koja omogućava pristup elementima, uz standardno po ključu, i po njihovu indeksu (kolekcija je poredana)
Stack	stog; kolekcija organizirana na FILO principu (*first-in*, *last-out*)

Tablica 5-6: ***Korisni tipovi kolekcija dostupnih unutar System.Collections***

Kolekcije vam pružaju baš to – omogućavaju vam stvaranje grupe objekata koja se dinamički puni i briše. Razlika se očituje i u tome što kolekcije neće posjedovati svojstvo *Length*, jer je njihova veličina promjenjiva, već će imati *Count*, jer on preciznije označava trenutnu količinu elemenata u kolekciji.

Kolekcije se nalaze unutar bazne klase *System.Collections,* koja pruža svu spomenutu funkcionalnost. U nastavku ćemo korištenje kolekcija objasniti na primjeru kolekcije *ArrayList*, no na raspolaganju vam stoji još nekoliko tipova kolekcija prikazanih u tablici 5-6, svaka sa svojom funkcionalnošću (osnovne metode korištene u sljedećim primjerima su iste), no njihovo proučavanje ostavljamo vama.

Sve ćemo objasniti na primjeru razreda. Napisat ćemo program koji će koristiti kolekciju *ArrayList* za praćenje učenika u razredu. Pretpostavka je, dakle, da na početku rada programa ne znamo koliko učenika ima razred – učenike ćemo upisivati i dodavati u kolekciju sve dok se ne upiše riječ "KRAJ".

```
System.Collections.ArrayList Razred = new System.Collections.ArrayList();
string ime;

while (true)
{
  Console.Write("Upišite ime učenika: ");
  ime = Console.ReadLine();
  if (ime.ToUpper() == "KRAJ") break;
  Razred.Add(ime);
}
```

Dakle, prvo smo deklarirali varijablu Razred, koja je tipa *System.Collections.ArrayList* odnosno kolekcija *ArrayList*. Koristimo i pomoćnu varijablu *ime,* u koju ćemo zapisivati upisana imena svih učenika razreda.

Definiciju kolekcije možete i pojednostaviti, tako da na početku programa napišete:

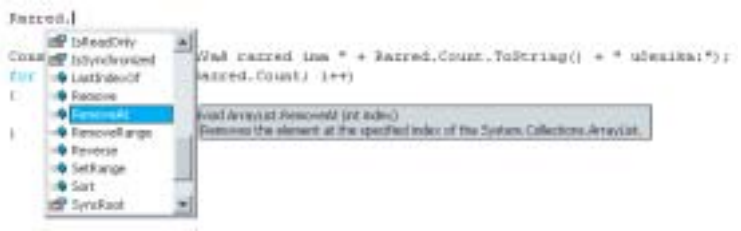
```
using System.Collections;
```

Sad novu kolekciju možete stvoriti na sljedeći način, bez nepotrebnog ponavljanja:

```
ArrayList Razred = new ArrayList();
```

Unos imena je riješen beskonačnom petljom. Unos se učitava u varijablu *ime*, a ukoliko je upisano "kraj" ili "KRAJ" (primijetite korištenje funkcije za pretvaranje u velika slova), petlja se prekida. U

početku programa kolekcija je prazna, a novi elementi se dodaju metodom *Add,* koja za parametar prima novi element.



Slika 5-5: ***ArrayList kolekcija skriva mnoštvo drugih metoda i svojstava, no ovdje smo objasnili samo najvažnije, koji je razlikuju od običnog polja.***

Uočite da kolekcija *ArrayList* nigdje nema definiran svoj tip. Naime, ona može primati bilo koji tip podataka zahvaljujući *boxingu* – proslijeđeni se podaci interno pretvaraju u tip *object*, pa je svejedno kojeg su tipa članovi kolekcije.

Da bismo saznali koliko članova ima kolekcija, iskoristit ćemo njeno svojstvo *Count*.

```
Console.WriteLine("Vaš razred ima " + Razred.Count.ToString() + " učenika.");
```

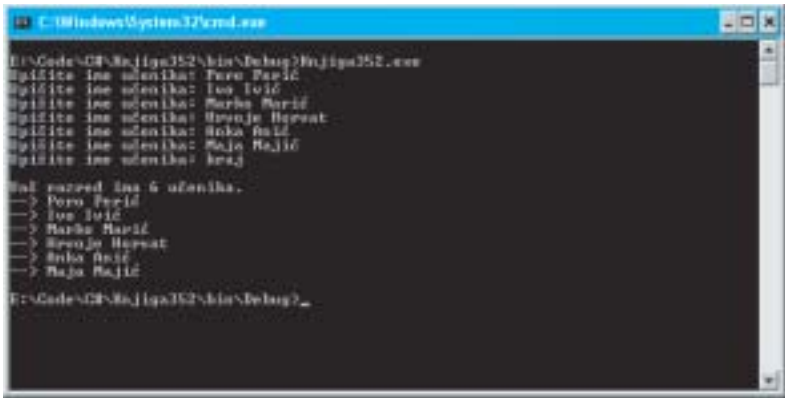
Pristup elementima kolekcije moguć je preko indeksa, baš kao i pri korištenju običnih polja. Sljedeći primjer ispisao bi sadržaj naše kolekcije.

```
for (int i = 0; i < Razred.Count; i++)
{
  Console.WriteLine(Razred[i]);
}
```

Pri korištenju kolekcija je najvažnije da uočite da kod njih ne trebamo znati koliko elemenata sadržavaju. Novi elementi dodaju se pomoću metode *Add*, a na isti način se mogu i micati. Želite li maknuti element s indeksom 0, napisat ćete:

```
Razred.RemoveAt(0);
```

Imajte na umu da će se cijela kolekcija tako smanjiti za jedan element: onaj koji je dosad imao indeks 1 sad će imati indeks 0, onaj s indeksom 2 dobit će indeks 1 i tako redom, jer je nulti izbrisan.



Slika 5-6: ***Primjer izvršavanja našeg programa koji radi s kolekcijom ArrayList***

Za sve članove...

Da biste se kretali kroz sve članove kolekcije ili polja, umjesto dosadašnje obične *for* petlje koja ide od člana s nultim indeksom do onog s indeksom za jedan manje od ukupnog broja članova, na raspolaganju vam stoji još jedan oblik *for*-petlje. Radi se o petlji *foreach* koja, kao što joj i ime kaže, prolazi kroz sve članove polja ili kolekcije. Pritom vi ne trebate eksplicitno zadati kroz koje sve članove ona treba proći, jer će proći kroz sve njih, ma koliko ih bilo.

```
int[] mojePolje = new int[] {1,
2, 3};
foreach (int clan in mojePolje)
{
  Console.WriteLine(clan);
}
```

Dakle, petlja *foreach* u varijablu *clan* učitava sve elemente polja (isto vrijedi i za kolekciju). Vrlo je važno da je varijabla u koju se učitavaju elementi istog tipa kao i svi članovi kolekcije kroz koju se prolazi. Ukoliko postoji mogućnost da se u kolekciji nalaze podaci različitih tipova, iskoristite dobri stari *boxing* – pretvorite sve članove u tip *object* i kasnije s njima radite što god želite. (Nemojte pretjerivati s korištenjem *boxinga*, jer to nije baš brza operacija.)

```
foreach (object clan in
mojaKolekcija)
{
  // ...
}
```

I na kraju, imajte na umu da petlja *foreach* ima nekoliko ograničenja. Prvo, unutar petlje *foreach* ne možete mijenjati vrijednost članova kolekcija ili polja, već ih možete samo čitati. Drugo, petlja *foreach* se ipak malo sporije izvršava od obične petlje *for*, zbog drugačijeg načina rada.

Klase i strukture

Postepeno se krećemo prema sve složenijim oblicima podataka – evo nas na klasama i strukturama! Razumijevanje objektno orijentiranog programiranja (koje se temelji na klasama) bit će objašnjeno u narednom poglavlju i nužno je za razumijevanje rada .NET-a, jer, kao što već znate, funkcionalnost .NET-a organizirana je u hijerarhiju klasa.

U nastavku ćemo objasniti kako možete stvarati svoje klase i čemu one uopće služe, te kako se koriste strukture (tip podataka veoma sličan klasama).

Klase

Na klase se može gledati kao na predloške za objekte. One opisuju funkcionalnost koje će pojedini objekti imati i podatke koje će sadržavati, no neće se raditi o konkretnom objektu. Primjer koji ćete susretati i u narednom poglavlju je *automobil* – klasa *Auto* može sadržavati opis objekta, njegove metode (primjerice, *Ubrzaj* i *Uspori*) te svojstva (*MaksimalnaBrzina*, *KolicinaGoriva* itd.).

No želite li stvoriti konkretan objekt koji će imati svoju vrijednost za maksimalnu brzinu i količinu goriva te koji će doista i obavljati akcije ubrzavanja i usporavanja, morat ćete stvoriti novi objekt iz klase *Auto*.

Evo kako biste definirali klasu *Auto*:

```
public class Auto
{
  // implementacija klase
}
```

Za definiranje klase koristi se ključna riječ *class* i ime klase, a unutar vitičastih zagrada se navodi njena implementacija. U implementaciji klase definiraju se članovi klase – radi se o svim metodama i svojstvima koje klasa ima.

Ključna riječ *public* trenutno nije bitna za razumijevanje klasa, a bit će objašnjena u narednom poglavlju.

Sve unutar vitičastih zagrada smatra se definicijom klase i tamo možete definirati njene članove. Na klase možete gledati kao na zasebne objekte koji u potpunosti sadržavaju svu svoju funkcionalnost i neovisni su o okolini. Tako svaka klasa može imati definirane metode i druge članove koji su nužni za njeno funkcioniranje. Definiranje članova unutar klasa ni po čemu se ne razlikuje od

standardnog definiranja članova u programu iz jednostavnog razloga – i na vaš program se gleda kao na zasebnu klasu.

Stoga ćemo definirati nekoliko osnovnih metoda i svojstava klase *Auto*:

```
public class Auto
{
  public int TrenutnaBrzina;

  public void Ubrzaj()
  {
   TrenutnaBrzina += 10;
  }

  public void Uspori()
  {
   TrenutnaBrzina -= 10;
  }

  public void Kreni()
  {
   while (TrenutnaBrzina != 50) Ubrzaj();
  }

  public void Stani()
  {
   while (TrenutnaBrzina != 0) Uspori();
  }
}
```

Naša klasa će imati jedno cjelobrojno svojstvo, koje će služiti za praćenje trenutne brzine. Definiramo ga kao i svaku drugu varijablu – jednostavno navodimo njen tip i ime.

U klasi su definirane i 4 metode – za ubrzavanje, usporavanje, pokretanje i zaustavljanje automobila. One interno koriste varijablu *TrenutnaBrzina* i međusobno se pozivaju da bi ispunile svoju funkcionalnost. Tako će metoda za ubrzavanje povećati brzinu za 10 km/h, dok će je metoda za usporavanje smanjiti za 10 km/h. Metoda za pokretanje automobila će pozivati metodu za ubrzavanje sve dok brzina automobila ne dosegne 50 km/h. Slično će i metoda za usporavanje automobila usporavati sve dok brzina ne dođe na nulu.

Stvaranje objekata iz klasa

Klase same za sebe ne vrijede mnogo. Kao što je rečeno, one služe kao predlošci za stvaranje objekata. Koristit ćemo ih poput bilo kojeg drugog tipa podataka, a za deklariranje ćemo koristiti samo ime klase.

```
Auto mojAuto;
```

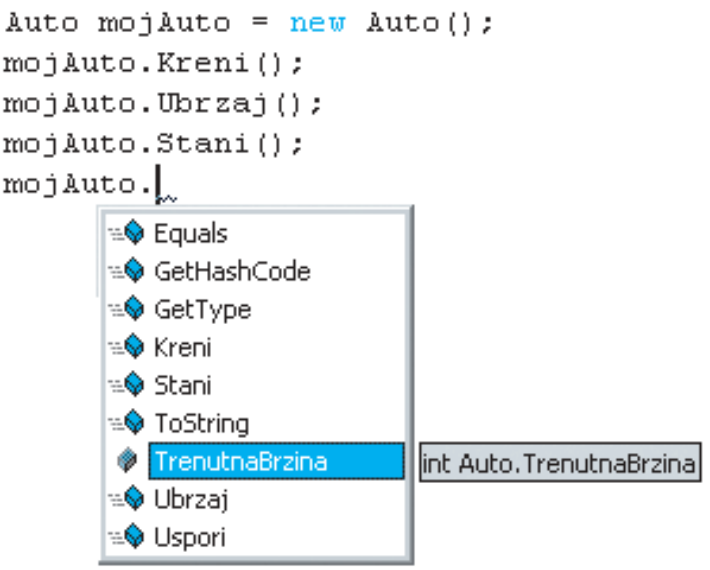
Prethodnom naredbom se deklarira novi objekt klase *Auto*, no on se još ne stvara u memoriji. Još uvijek ga nije moguće koristiti, jer smo zasad tek *najavili* njegovo postojanje. Želite li ga stvoriti, iskoristit ćete naredbu *new*:

```
mojAuto = new Auto();
```

Tek sada je stvoren novi objekt klase *Auto,* koji sadrži svu njenu funkcionalnost (metode za ubrzavanje, usporavanje itd.). Naravno, kao i kod običnih varijabli, sve ste to mogli izvesti u jednoj naredbi:

```
Auto mojAuto = new Auto();
```

Sad možete pozivati sve naredbe i koristiti vrijednosti svih varijabli iz objekta. Na slici 5-7 prikazana je IntelliSense pomoć Visual Studia koja za objekt klase *Auto* ispisuje sve mu dostupne metode i članove.



Slika 5-7: ***IntelliSense će ispisati sve dostupne članove objekta.***

Da biste koristili članove nekog objekta, navedite ih odvojene točkom od imena objekta.

Sve metode možete koristiti u svom programu. Slijedi jednostavan primjer u kojem simuliramo vožnju:

```
Auto mojAuto = new Auto();
mojAuto.Kreni();
mojAuto.Ubrzaj();
Console.WriteLine(mojAuto.TrenutnaBrzina); // 60
mojAuto.Stani();
```

Dakle, prvo smo stvorili novi objekt klase *Auto*, a zatim smo pozivali njegove metode. Sve te metode su definirane u samoj klasi, pa ih može koristiti svaki objekt te klase. Tako pokrećemo automobil, ubrzavamo ga, ispisujemo trenutnu brzinu (u tom trenutku će biti 60 km/h) i na kraju ga zaustavljamo.

Konstruktori i destruktori

Klase imaju i još jednu veoma korisnu mogućnost – *konstruktore* i *destruktore*. Radi se o metodama koje se pozivaju pri stvaranju objekta iz neke klase (tome služi *konstruktor*) i pri uništavanju odnosno micanju objekta iz memorije (*destruktor*).

Da bismo uočili kako rade i u kojem se trenutku pozivaju konstruktori i destruktori, proširit ćemo prethodni primjer. Pri stvaranju objekta klase *Auto* želimo postaviti trenutnu brzinu na 5 km/h – recimo da želimo da novi automobili automatski voze malom brzinom. Tome će nam poslužiti konstruktor metoda.

Zbog jednostavnosti ćemo pri uništavanju objekta odnosno u destruktor metodi samo ispisati poruku – u složenijim primjerima destruktor će služiti za brisanje referenci, micanje nekih dodatnih objekata i slične operacije.

```
public class Auto
{
  public Auto()
  {
   Console.WriteLine("Stvaram objekt klase Auto.");
   TrenutnaBrzina = 5;
  }

  ~Auto()
  {
   Console.WriteLine("Brišem objekt klase Auto.");
  }

  // ostatak implementacije...
```

```
}
```

Konstruktor i destruktor imaju isto ime kao i klasa u kojoj se nalaze. Dok konstruktor ima ispred sebe ključnu riječ *public* i nema oznaku tipa (jer ne vraća nijednu vrijednost, niti je moguće metodu konstruktora pozivati iz kôda – automatski se poziva pri stvaranju objekta), destruktor nema ni ključnu riječ *public* ni oznaku tipa, već samo znak "~" ispred imena.

U konstruktor tako možete ubaciti bilo koji kôd koji želite da se izvrši pri stvaranju objekta, a u destruktor sve što želite izvršiti po uništavanu samog objekta. Evo i detaljnije objašnjenog kôda primjera sa stvaranjem i korištenjem objekta:

```
static void Main(string[] args)
{

  // Deklaracija objekta klase Auto
  Auto mojAuto;

  // Stvaranje novog objekta – pozivanje konstruktora!
  mojAuto = new Auto();

  // Korištenje metoda objekta
  mojAuto.Kreni();
  mojAuto.Stani();

} // Uništavanje objekta – pozivanje destruktora!
```

Zbog Garbage Collectora – sustava u .NET-u koji prati korištenje objekata i sam automatski uništava nekorištene objekte – vrlo je teško odrediti točan trenutak kad se neki objekt uništava. U jednostavnim aplikacijama u konzoli to se najčešće dešava na kraju izvršavanja programa, no u složenijim aplikacijama je to vrlo teško odrediti, pa se stoga ne preporučuje oslanjanje na destruktor za izvršavanje kritičnih operacija.

Ispis takvog programa bit će doista jednostavan te samo potvrđuje komentare iz prethodnog primjera i dokazuje da su se konstruktor i destruktor doista izvršili.

```
Stvaram objekt klase Auto.
Brišem objekt klase Auto.
```

No konstruktorima se mogu prosljeđivati i parametri. Možda nećete biti zadovoljni postavljanjem početne brzine na 5 km/h, već ćete to htjeti odrediti za svaki objekt posebno. Izrada konstruktora koji prima parametar jednostavan je i svodi se na izradu metode koja prima parametar:

```
public class Auto
{
  public Auto(int PocetnaBrzina)
  {
   TrenutnaBrzina = PocetnaBrzina;
  }

  // ostatak implementacije...

}
```

Kao što vidite, konstruktor očekuje cjelobrojni parametar, koji se upisuje u varijablu *TrenutnaBrzina*. Sad je i stvaranje objekta klase *Auto* malo drugačije, no pruža veće mogućnosti:

```
Auto mojSporiAuto = new Auto(5);
Auto mojBrziAuto = new Auto(100);

// ispisuje 5
Console.WriteLine(mojSporiAuto.TrenutnaBrzina);

// ispisuje 100
Console.WriteLine(mojBrziAuto.TrenutnaBrzina);
```

Unutar zagrada pri stvaranju objekta navode se parametri za konstruktor. Uočite da u prethodnim primjerima unutar tih zagrada nije postojalo ništa, jer konstruktor nije ni primao parametre. Kako smo napisali konstruktor koji prima jedan parametar, nužno ga je navesti pri stvaranju objekta.

Vrlo se lako može provjeriti da li konstruktor doista radi – stvaramo dva objekta kojima su proslijeđene različite vrijednosti. Tako će objekt *mojSporiAuto* za trenutnu brzinu imati vrijednost 5 km/h, a *mojBrziAuto* 100 km/h, jer su to vrijednosti proslijeđene konstruktorima pri stvaranju.

Kad svladate osnove objektno orijentiranog programiranja (naredno poglavlje), naučit ćete napraviti tzv. preopterećene metode. Tada ćete moći napraviti objekte koji mogu imati dva konstruktora – jedan koji ne prima niti jedan parametar (i radi kao u prvim primjerima), i drugi koji prima parametar početne brzine (i radi kao u prethodnom primjeru). Takve objekte ćete zato moći stvarati na dva načina: s parametrom i bez njega.

Strukture

Funkcionalnost struktura veoma je slična funkcionalnosti klasa, jer se i na strukture može gledati kao na zasebne elemente koji u sebi sadržavaju svu funkcionalnost. No razlika ipak postoji. Dok su klase zapravo reference (stvaranjem s *new* se tek zauzima mjesto u memoriji, a sama varijabla objekta samo ukazuje na tu memorijsku lokaciju), strukture su vrijednosni tip podataka. Usto, strukture, za razliku od klasa, nije moguće nasljeđivati.

Dakle, strukture mogu implementirati sve članove kao i klase, a za definiranje strukture se koristi ključna riječ *struct*.

```
public struct Tocka
{
  public int x, y;

  public void PomakniTocku(int PomakX, int PomakY)
  {
   x += PomakX;
   y += PomakY;
  }
}
```

Kao što vidite, načinjena je struktura koja predstavlja neku točku – koristi dvije cjelobrojne vrijednosti za pohranjivanje koordinata *x* i *y* te jednu metodu za pomak točke.

```
// Deklaracija strukture
Tocka A = new Tocka();

// Postavljanje varijabli
A.x = 50;
A.y = 100;

// Pozivanje metoda
A.PomakniTocku(-10, -50);
```

I strukture mogu imati konstruktore koji se definiraju isto kao i kod klasa (moraju biti istog imena kao i struktura) – u tom slučaju nove strukture stvarate baš kao i klase, a konstruktoru možete proslijediti i parametre.

```
Tocka A = new Tocka(3, 4);
```

Pripazite: strukture ne mogu imati definirane destruktore.

Strukture ili klase?

Na prvi pogled su strukture i klase identične – obje mogu sadržavati članove, obje mogu imati konstruktore i, kao i svi drugi tipovi podataka u .NET-u, izvedene su iz glavnog objekta *Object*. No razlika postoji: dok su klase reference, strukture su vrijednosni tip podataka i njihovi se podaci spremaju na stogu, u glavnom dijelu memorije.

Pristup stogu, gdje se spremaju strukture, veoma je brz i lagan, no ukoliko na njega spremite velike količine podataka, to će bitno utjecati na performanse aplikacije. To znači da su strukture idealne za male objekte koji sadržavaju male količine podataka. Klase su, s druge strane, prikladnije za veće objekte, za koje se očekuje da će duže postojati u memoriji (pa ih se stoga ne želi smještati na stog) i koji sadržavaju veće količine podataka.

Želite li tako stvoriti polje objekata, za primjer uzmimo polje od 10 točaka koje zajedno čine pravac. Pritom su definirane klasa i struktura za točku:

```
public struct TockaStruktura
{
  // implementacija kao u prethodnom primjeru
}

public class TockaKlasa
{
  // implementacija identična strukturi
}
```

Sljedeći program bi tako stvorio i inicijalizirao dva polja od 10 točaka, jedno koje koristi strukture za zapis pojedinih točaka, i drugo koje koristi klase. Primijetite da je stvaranje polja struktura ili klasa po sintaksi identično stvaranju polja s vrijednostima tipa *int* ili *string*.

```
TockaStruktura[] PravacStruktura = new TockaStruktura[10];
TockaKlasa[] PravacKlasa = new TockaKlasa[10];
```

Na ovom se primjeru može uočiti bitna razlika u načinu rada struktura i klasa. Ukoliko koristite polje klasa, u memoriji će se stvoriti 11 objekata – jedan objekt za polje i po jedan objekt za svaki od 10 članova polja. To je tako zato što će članovi polja biti zapravo reference na same objekte spremljene negdje drugdje u memoriji.

S druge strane, korištenjem polja struktura, u memoriji će se stvoriti samo jedan objekt, i to onaj za polje. Sve strukture točaka bit će spremljene unutar polja, jer se radi o vrijednosnom tipu podataka.

No takav način rada struktura nije nužno bolji. Već je prije spomenuto da korištenjem struktura možete usporiti aplikaciju – ukoliko svojim metodama prosljeđujete strukturu kao parametar, ona

će morati biti kopirana i metoda će koristiti novu instancu strukture. To je posljedica vrijednosnog tipa podatka – metodi se ne može proslijediti referenca na već postojeću instancu u memoriji, već se treba raditi nova kopija samih vrijednosti podataka. Time se, dakako, zauzima dodatna memorija.

Prije no što se odlučite na korištenje struktura ili klasa, dobro razmislite za što ćete takve objekte koristiti. Treba imati na umu i da strukture nije moguće nasljeđivati, pa one nisu izbor kad želite napraviti pravu objektno orijentiranu aplikaciju i razbiti svu funkcionalnost aplikacije na hijerarhiju objekata. (Više o tome u narednom poglavlju!) Pravilnim izborom i korištenjem možete ubrzati rad svoje aplikacije, a to je nešto oko čega se vrijedi potruditi.

Delegati

U .NET-u postoji još jedan način pozivanja metoda. Radi se o *delegatima*, koji zapravo predstavljaju pokazivače na metodu. Pri deklaraciji delegata definirate tip metode koju može pozivati (tip podataka koju vraća) i parametre (sve to zajedno čini *potpis* metode).

Važno je shvatiti osnovni koncept delegata. Na njih se može gledati kao na poseban tip podataka, koji u sebi sadržava poziv na neku metodu. Tako jedan delegat može biti iskorišten za pozivanje više različitih metoda s istim potpisom (isti povratni tip, parametri i prava pristupa).

Dakle, nužno je definirati tip metoda koji se može pozivati korištenjem delegata. Sljedeći primjer omogućava pozivanje metoda koje vraćaju *double* vrijednost, a ujedno i primaju *double* vrijednost.

```
public delegate double mojDelegat (double x);
```

Delegati predstavljaju dio klase i stoga moraju biti definirani izvan bilo koje metode.

Da bismo mogli iskoristiti definiranog delegata, napravit ćemo dvije metode s istim potpisom. One će biti krajnje jednostavne i služit će isključivo za pokazivanje primjera.

```
public double kvadrat(double x) { return Math.Pow(x, 2); }
public double korijen(double x) { return Math.Sqrt(x); }
```

Krenimo sad s uporabom delegata. Da bismo mogli pozvati metodu putem delegata, stvorit ćemo njegovu novu instancu, koja će sadržavati pokazivač na željenu metodu. Evo kako bi izgledali delegati za poziv prethodne dvije metode:

```
mojDelegat dKvadrat = new mojDelegat(kvadrat);
mojDelegat dKorijen = new mojDelegat(korijen);
```

Dakle, definirali smo dvije varijable tipa *mojDelegat*. Njihove smo instance stvorili korištenjem operatora *new*, a unutar zagrada smo naveli metodu koju će svaki od delegata pozivati. Kako je delegat definiran tako da prima i vraća *double* vrijednost, na nove instance *dKvadrat* i *dKorijen* može se gledati kao na takve metode.

Korištenje delegata očito je na sljedećim primjerima. Iskoristit ćemo *dKvadrat* i *dKorijen* za poziv metoda.

```
double a = dKvadrat(5);  // 25
double b = dKorijen(49); // 7
```

Kao što vidite, korištenje delegata identično je pozivima običnih metoda. Tako u dvije varijable *double* tipa *a* i *b* spremamo kvadrat ili korijen brojeva. Imajte na umu da delegate možete koristiti za poziv bilo kakvih metoda, svejedno da li vraćaju neke podatke ili ne.

Iako je dosad pokazano da su delegati jednostavni za korištenje, možda ste si ipak postavili pitanje zašto ih uopće koristiti kad je jednostavnije direktno pozivati metode. Ne brinite, na vaše pitanje bit će odgovoreno kad se pozabavimo događajima u aplikacijama i asinkronim pozivima web-servisa. Zasad je važno da shvatite njihov koncept i način korištenja.

6. POGLAVLJE

Objektno orijentirano programiranje

U ovom poglavlju:

- Osnovni koncepti objektno orijentiranog programiranja
- Što je to učahurivanje, višeobličje, preopterećenje, prekoračenje i nasljeđivanje
- Određivanje prava pristupa članovima klasa
- Razlika između klasa, apstraktnih klasa i sučelja
- Kako nasljeđivati klase i nadograditi im funkcionalnost
- Kako implementirati sučelja

Koncept objektno orijentiranog programiranja najvažniji je koncept u .NET-u. Da biste mogli u potpunosti iskoristiti mogućnosti .NET-a, morat ćete naučiti objektno programirati. Ukoliko ste dosad programirali u jezicima kao što su C, Pascal ili Visual Basic, niste se još upoznali s tim konceptom – svu funkcionalnost programa sadržavale su posebne metode, a vi biste ih pozivali po potrebi. Koncept OOP-a je drukčiji i, kao što mu samo ime kaže, zasniva se na objektima i njihovim odnosima. U ovom poglavlju

upoznat ćete osnovne pojmove OOP-a koji će vam trebati i pri najosnovnijem .NET programiranju, stoga krenimo na posao.

Kad se govori o objektno orijentiranom programiranju u .NET-u, misli se na to da je svaki od .NET programskih jezika objektno orijentiran, a razlika postoji samo u sintaksi. Kako bi bilo preopširno detaljno obrađivanje mogućnosti svakog jezika, kao i u ostatku knjige, posvetit ćemo se C#-u.

Osnovni pojmovi OOP-a

Glavna stvar u objektno orijentiranom programiranju su *objekti*. Na objekte možete gledati kao na zasebne strukture koje sadržavaju neke povezane podatke i metode. Primjerice, radite li program koji simulira vožnju automobila, stvorit ćete poseban objekt za svaki auto. Taj objekt će sadržavati osnovne podatke, kao što su boja automobila, količina benzina u spremniku, njegova maksimalna brzina, broj osoba koje može prevoziti. Sadržavat će i neke metode: za ubrzavanje, usporavanje, zaustavljanje, skretanje i slično.

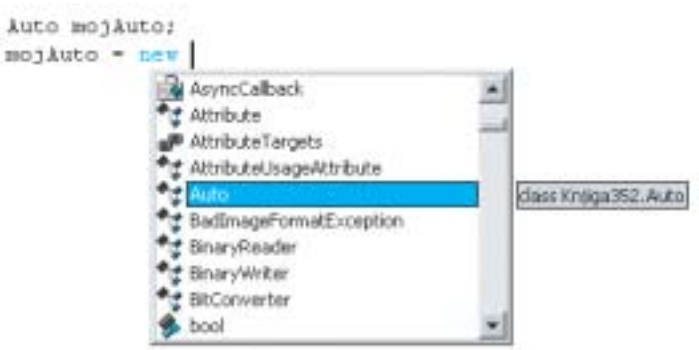
Ukoliko u svom programu želite simulirati neki automobil, jednostavno ćete stvoriti novi objekt te pozivati njegove metode za upravljanje vozilom, a nećete se morati brinuti o njihovoj funkcionalnosti, jer će ona biti u objektu. No ne mora kompletna funkcionalnost nekog objekta biti dostupna korištenju. Primjerice, skoro nikad nećete pri simuliranju vozila pozivati metodu za okretanje ključa u bravi, za prebacivanje u nultu brzinu, za paljenje svjetala i slično, već ćete to ostaviti metodi za pokretanje automobila. Ona će biti zadužena za pokretanje svih tih akcija, jer se to od nje i očekuje. Te druge akcije su, dakle, vama skrivene i koriste se samo interno unutar objekta.

Klase i objekti

O klasama je već bilo riječi u prethodnim poglavljima, a one su ključne za razumijevanje i korištenje OOP-a u bilo kojem objektno orijentiranom jeziku, poput C++-a ili Delphija, a ne samo u .NET jezicima. Na klase se može gledati kao na predloške ili nacrte za objekte. U njima su točno definirani svi članovi objekta, njihova kompletna funkcionalnost, a postavljene su i inicijalne vrijednosti objekta potrebne za ispravan rad.

Pritom je vrlo važno upamtiti da se definiranjem klasa ne zauzima memorija, jer one služe samo kao nacrt. Tek se njihovim instanciranjem zauzima memorija, a ta stvorena instanca se naziva "objekt". Samo za podsjetnik, novi objekt se stvara na sljedeći način (pretpostavimo da je prije definirana klasa *Auto*):

```
// Deklaracija varijable tipa "Auto", ne zauzima se memorija
Auto mojAuto;

// Instanciranje novog objekta, zauzima se memorija
mojAuto = new Auto();
```



Slika 6-1: ***Visual Studio automatski će vam pri stvaranju novog objekta ponuditi odgovarajuću klasu i tako olakšati pisanje kôda.***

Svaki objekt je u potpunosti funkcionalna jedinica i sadrži sve potrebne podatke i varijable za svoj ispravan rad te otkriva svoju funkcionalnost *vanjskom svijetu*. Objekti su u stvarnom svijetu svuda oko nas – to su već spomenuti automobili, računala, instrumenti itd. Predstavljanje nekog objekta iz stvarnog svijeta objektom u programskom jeziku naziva se *apstrakcija*. Postupak pojednostavljenja objekata ključni je koncept programiranja.

Objektni modeli

Jednostavni objekti mogu sadržavati samo nekoliko varijabli, metoda ili događaja. Složeniji objekti mogu sadržavati čitav niz varijabli, gomile metoda, a može se pojaviti i potreba za podobjektima. Naprimjer, automobil: on može sadržavati poseban objekt *Motor*, četiri objekta *Kotač* i slično. Kompozicija svih tih objekata čini objekt *Auto,* pa tako u slučaju da objekt *Motor* ima vrijednost atributa *Cilindri* 4, takav automobil će imati drugačije performanse nego objekt čiji podobjekt *Motor* ima za vrijednost atributa *Cilindri* 8. Tu cijela stvar ne prestaje – podobjekti mogu imati svoje podobjekte i tako u nedogled.

Hijerarhija takvih objekata naziva se *objektni model* i predstavlja strukturu i međusobne odnose povezanih objekata.

U slučaju da neki jezik podržava rad s objektima, nemojte da vas to prevari – to još uvijek ne znači da on podržava objektno orijentirano programiranje. Evo i ukratko mogućnosti koje jezik mora podržavati da bi bio objektno orijentiran:
- Mora podržavati objekte koji predstavljaju apstrakciju stvarnog svijeta. Oni moraju imati mogućnost spremanja i skrivanja svog stanja te sučelje (izložene metode) koje definira operacije nad objektom.
- Svi objekti moraju pripadati nekoj klasi.
- Treba biti podržano nasljeđivanje između klasa.

Učahurivanje ili enkapsulacija

Jedan od glavnih principa OOP-a je učahurivanje (engl. *encapsulation,* enkapsulacija). Da bismo ga objasnili, potrebno je vratiti se na opis neke klase. On se sastoji od dva dijela: sučelja (predstavlja pogled na neku klasu iz *vanjskog svijeta*) i implementacije (ugradnja mehanizama pomoću kojih se ispunjava funkcionalnost objekata opisana u sučelju). Koncept učahurivanja govori da je implementacija objekta potpuno neovisna o njegovu sučelju.

Aplikacija koja koristi vaš objekt komunicira s njime preko vidljivog joj sučelja. Sjetimo se primjera automobila – klasa *Auto* ima sučelje koje se sastoji od metoda za ubrzavanje, usporavanje, zaustavljanje i skretanje, a implementacija se nalazi u skrivenim objektima kao što su oni za okretanje ključa u bravi, paljenje svjetala i slično. Sve dok je sučelje isto, aplikacija može ispravno komunicirati i upravljati objektom, koliko god se implementacija mijenjala. Tako naknadno možete potpuno promijeniti funkcionalnost metode za paljenje svjetala ili je čak izbrisati, a aplikacija će dalje ispravno raditi, jer će pozivati metodu za pokretanje automobila koja i dalje postoji u sučelju.

Učahurivanje je, dakle, koncept koji kaže da ništa ne smije ovisiti o unutrašnjim detaljima nekog objekta. Objekti trebaju međusobno komunicirati isključivo preko javno dostupnih i vidljivih metoda i svojstava. Kasnije će u poglavlju biti detaljnije objašnjeno kako se postiže učahurivanje odnosno kako se skriva implementacija od *vanjskog svijeta*.

Višeobličje ili polimorfizam

Višeobličje (engl. *polymorphism,* polimorfizam) jest mogućnost da klase odnosno objekti na različite načine implementiraju ista sučelja. Drugim riječima, višeobličje omogućava pozivanje istih metoda i korištenje istih svojstava bez obzira na to o kojoj se implementaciji sučelja radi. Objasnimo sve na konkretnom primjeru, naravno, s automobilima. Recimo da vaša aplikacija komunicira s objektom *Auto* koji ima već opisano sučelje (ubrzavanje, usporavanje, zaustavljanje, skretanje). Ukoliko po-

stoji još objekata, kao što su *Kamion* ili *Brod*, koji imaju identično sučelje, s njima možete komunicirati na isti način, bez pridavanja posebne pozornosti tome o kojem se objektu radi. Naravno, kasnije u poglavlju će sve biti objašnjeno na primjerima.

Višeobličje sučelja

Ukoliko klasa u sebi ne sadrži implementaciju i konkretan kôd, već samo definiciju sučelja kao opis njenih mogućnosti, tada ju jednostavno nazivamo sučeljem (engl. *interface*). Sučelje, dakle, definira kakvu funkcionalnost treba imati neka klasa, no ne petlja se u njenu implementaciju.

Neki objekt može podržavati više sučelja, što znači da ima višestruku funkcionalnost. S njime je moguće komunicirati preko bilo kojeg od podržanih sučelja. Isto tako, više različitih objekata može podržavati isto sučelje, baš kao u prethodnom primjeru s automobilima.

Dakle, postoji sučelje *IVozilo* (prefiks *I* dolazi od *interface;* prema konvenciji, njime se uvijek označava sučelje) koje definira metode za ubrzavanje, usporavanje, zaustavljanje i skretanje. Ono ne implementira te metode, već isključivo navodi da ih podržava. Naša klasa *Auto* implementira sučelje *IVozilo* tako da ugrađuje funkcionalnost u obavezne metode. No isto tako ima smisla da sučelje *IVozilo* implementiraju i klase *Kamion* i *Brod*. S njima je, dakle, moguće komunicirati na isti način i nije uopće bitno o kojoj se klasi radi – kako sve one implementiraju isto sučelje, moguće je nad njima koristiti bilo koju metodu tog sučelja.

Nasljeđivanje

Koncept nasljeđivanja (engl. *inheritance*) dopušta da ugradite svu funkcionalnost već postojeće klase u neku novu klasu. Nasljeđivanjem nova klasa preuzima svu funkcionalnost od one koju je naslijedila – sve njene metode, varijable i druge članove. No novu klasu možete mijenjati i dodati joj novu funkcionalnost, a isto tako, što je mnogo važnije, možete *prekoračiti* postojeće nasljeđene metode i napisati potpuno nove s drugačijom funkcionalnošću.

Vratimo se opet na primjer automobila. Možete tako napraviti novu klasu *SportskiAuto* koja glavnu funkcionalnost nasljeđuje od klase *Auto* – sve njene metode i ostale članove. Na vama je samo da implementirate nove metode, primjerice za spuštanje pomičnog krova, a možete promijeniti i postojeće, primjerice napisati drugačiju metodu za zaustavljanje da više koristi kočnice i tako naglo koči.

Klasa u .NET-u može nasljeđivati od samo jedne klase, koja se tad naziva *baznom* klasom. No primijetite da zato klasa može nasljeđivati od više sučelja (naravno, pritom ima obavezu implementirati sve metode svih naslijeđenih sučelja).

Pristup članovima klase

Za izvedbu učahurivanja podataka i metode klasa te za otkrivanje funkcionalnosti nekog objekta *vanjskom svijetu* ključno je određivanje prava pristupa pojedinim članovima klase. U tablici 6-1 nalaze se ključne riječi za određivanje prava pristupa.

Tablica 6-1: ***Ključne riječi za određivanje prava pristupa članovima klasa u C#-u***

Ključna riječ	Utjecaj na člana
public	Član je dostupan svima
private	Član je dostupan samo unutar klase
internal	Članu se može pristupiti iz svih klasa unutar istog *assemblyja*
protected	Članu se može pristupiti samo iz klase koja ga definira i iz klasa koje od nje nasljeđuju
protected internal	Predstavlja uniju *protected* i *internal* ključnih riječi – članu je moguće pristupiti iz svih klasa unutar istog *assemblyja* te iz svih klasa koje nasljeđuju u klasu u kojoj je član definiran

Član deklariran s *public* ključnom riječi vidljiv je kompletnom kôdu izvan klase. Tako je moguće pozivati sve *public* metode neke klase i mijenjati sve *public* varijable od bilo kuda iz kôda. Dakle, sve metode deklarirane s *public* su dio sučelja i pomoću njih je moguće komunicirati s objektom. Članovi deklarirani s *private* vidljivi su samo unutar te klase i najčešće predstavljaju implementaciju, nevidljivu vanjskom svijetu. Vrlo je važna i *protected* ključna riječ, kojom dozvoljavamo pristup članu samo iz klase u kojoj je definiran i iz svih klasa koje ju nasljeđuju.

Pod *članovima* neke klase podrazumijevaju se sve njene metode, varijable i događaji koji pripadaju toj klasi.

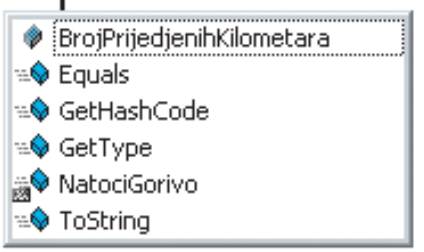
Tablica 6-2:
Pojednostavljena tablica 6-1, u kojoj je zorno prikazano u kojem je dijelu kôda dostupan član označen nekom ključnom riječi

Ključna riječ	U klasi u kojoj je definiran	U klasama koje ga nasljeđuju	U svim klasama istog *assemblyja*	Kodu izvan istog *assemblyja*
public	✔	✔	✔	✔
private	✔			
internal	✔		✔	
protected	✔	✔		
protected internal	✔	✔	✔	

Evo i konkretnog primjera kako se definiraju prava pristupa članovima neke klase.

```
public class Auto
{
    // Varijabli je dozvoljen bezuvjetan pristup
    public int BrojPrijedjenihKilometara;

    // Metodu mogu pozivati sve metode iz ove klase
    // i assemblyja, ali ne i vanjskog koda
    internal void NatociGorivo()
    {
    }

    // Varijabli mogu pristupati samo članovi ove klase
    private int TrenutnaBrzina;

    // Metodu mogu pozivati sve metode ove klase
    // i svih klasa koje ju nasljeđuju  u
    protected void UpaliSvjetlo()
    {
    }
}
```

```
Auto mojAuto;
mojAuto = new Auto();

mojAuto.
```



Slika 6-2:
Pri pisanju kôda automatski ćete dobiti popis svih članova kojima možete pristupiti – na popisu je vidljiva varijabla BrojPrijedjenihKilometara (public) i NatociGorivo (internal).

Ukoliko u C#-u ne navedete ključnu riječ pristupa pri deklaraciji člana, njemu će pristup biti dozvoljen pod *private* uvjetima odnosno moći će mu se pristupiti samo unutar njegove klase.

I klasama pristupaju, zar ne?

Vjerojatno ste primijetili da u prethodnom primjeru i klasa ima oznaku prava pristupa. Kod njih je situacija vrlo slična kao i kod samih članova klasa, samo na drugoj razini. Primjerice, *public* klase mogu se instancirati iz bilo kojeg objekta u aplikaciji (što je i *defaultno* ponašanje, ukoliko se ne navede pravo pristupa), dok su *internal* klase dostupne samo unutar istog *assemblyja*.

Zajednički *(static)* članovi

Dosad smo se susreli samo s *običnim* članovima klasa, tj. onima koji su jedinstveni za svaku instancu objekta. Primjerice, koristi li klasa varijablu *Broj*, svaki objekt te klase moći će imati spremljenu drugu vrijednost u tu varijablu.

```
PrimjerKlasa objekt1 = new PrimjerKlasa();
PrimjerKlasa objekt2 = new PrimjerKlasa();
objekt1.Broj = 10;
objekt2.Broj = 73;
```

Istovremeno su spremljene dvije vrijednosti odnosno svaki objekt ima spremljenu vlastitu vrijednost varijable *Broj*.

No moguće je i drugačije ponašanje – definiranje posebnih, zajedničkih varijabli. Te varijable će, dakle, biti zajedničke za sve objekte neke klase. Primjerice, da smo definirali da klasa ima zajedničku varijablu *Broj*, postojala bi samo jedna njena vrijednost za sve instance (a ne dvije, kao u gornjem primjeru, ili više).

Zajednička varijabla se definira korištenjem *static* ključne riječi:

```
public class PrimjerKlasa
{
    public static int Broj;

    // ...
}
```

Varijablu *Broj* i dalje je moguće koristiti u svim metodama klase, no postojat će samo jedna njena vrijednost, zajednička za sve instance.

Primijetite da *static* ključna riječ ne određuje prava pristupa, već samo je li član zajednički za cijelu klasu. Svi *static* članovi i dalje mogu biti *public*, *private* itd.

No da biste pristupili varijablama koje su zajedničke za cijelu klasu, morat ćete malo promijeniti sintaksu. Umjesto pristupa varijabli preko instance objekta, kao u prethodnim primjerima, za pristup ćete morati koristiti ime klase.

```
PrimjerKlasa objekt1 = new PrimjerKlasa();
PrimjerKlasa.Broj = 10;
```

Netočna sintaksa bi bila da pokušate promijeniti sadržaj zajedničke varijable naredbom objekt1.Broj = ...".

Primijetite i da metode mogu biti zajedničke za cijelu klasu, tj. za sve njene objekte. No kako one u tom slučaju ne pripadaju nekom objektu, ne mogu koristiti njegove varijable – u zajedničkim

metodama možete koristiti samo zajedničke varijable i, naravno, varijable prenesene metodama preko parametara ili deklarirane unutar same metode.

Kako zajedničke varijable pripadaju klasi, a ne nekom njenom objektu, njima je moguće pristupati čak i prije nego što instancirate neki objekt te klase. Konkretno, u gornjem primjeru niste trebali stvoriti objekt *objekt1* prije no što ste upisali 10 u varijablu *Broj*. Ukoliko pak neki objekt klase sa zajedničkom varijablom stvorite nakon što već toj varijabli pridijelite vrijednost, iz tog objekta ćete joj svejedno moći pristupati, jer je ona vezana za klasu.

Preopterećenje

Preopterećenje (engl. *overloading*) jedan je od ključnih koncepata objektno orijentiranog programiranja. Naime, iza te pomalo nejasne riječi skriva se vrlo korisna mogućnost – u .NET-u možete stvarati više članova neke klase koji imaju isto ime, no koriste se za različite stvari. Preopterećenje se najčešće koristi pri pisanju metoda. Primjerice, možete stvoriti dvije metode imena *Ispis*, od kojih jedna može imati samo jedan parametar, i to tekst koji treba ispisati, a druga metoda može imati dva parametra, s tim da drugi određuje koliko puta tekst treba ispisati.

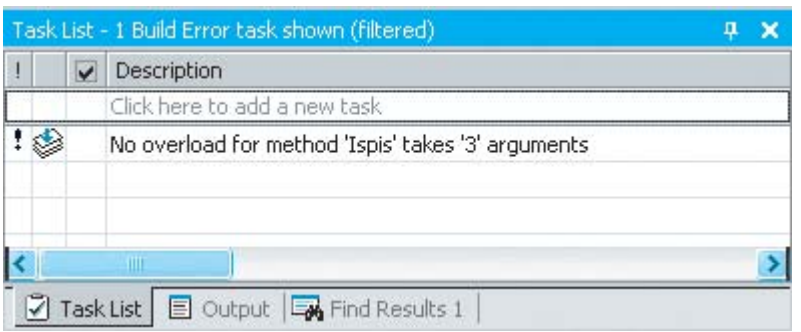
Ključ je, dakle, u različitim parametrima koje primaju te istoimene funkcije. Evo kako biste izveli prethodni primjer:

```
public void Ispis(string tekst)
{
  Console.WriteLine(tekst);
}

public void Ispis(string tekst, int broj)
{
   for (int i = 0; i < broj; i++)
      Console.WriteLine(tekst);
}
```

Dakle, preopterećene metode su metode s istim imenom, no s drugačijim parametrima. Parametri se mogu razlikovati u broju (konkretno, naš primjer ima jednu metodu s jednim parametrom, a drugu metodu s dva parametra) ili u svom tipu (primjerice, mogu obje metode imati samo jedan

parametar, no njegov se tip može razlikovati od metode do metode, pa tako može postojati metoda koja radi isključivo s brojevima, dok druga radi isključivo s tekstom).

Slika 6-3:
Ukoliko metodu Ispis pokušate pozvati s 3 parametra, javit će se pogreška, jer takva preopterećena metoda ne postoji.



Iako, dakle, preopterećene metode moraju imati drugačiji *potpis* (parametre), ne moraju sve vraćati istu vrijednost ili pak imati ista prava pristupa (jedna može biti *public*, druga *private* itd.). Pri izvršavanju programa i pozivu neke preopterećene metode provjeravaju se tipovi podataka koji su proslijeđeni pozvanoj metodi. Ukoliko se pak ne pronađe metoda koja prima poslane tipove, javlja se greška.

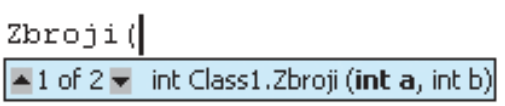
```
// Poziva se prva metoda
Ispis("Dobar dan!");

// Poziva se druga metoda
Ispis("Dobar dan!", 10);
```

Kao što je spomenuto u tekstu, preopterećenja metoda idu dalje od istoimenih metoda s različitim brojem parametara – češće je preopterećenje metoda s istim brojem parametara, no različitih tipova. To se spominje jer VB.NET omogućava definiranje *opcionalnih* parametara čije navođenje nije obavezno. Da smo naš primjer pisali u VB.NET-u, ne biste se morali brinuti preopterećenjem metoda, već biste jednostavno drugi parametar metode učinili opcionalnim i, ukoliko on ne bi bio naveden pri pozivu funkcije, smatralo bi se da je broj ispisivanja jednak 1.

Dakle, vrlo je koristan i primjer u kojem postoje preopterećene metode s istim brojem parametara, no različitih tipova. Evo i primjera:

```
public int Zbroji(int a, int b)
{
    return a + b;
}

public string Zbroji(string a, string b)
{
    return a + " " + b;
}
```

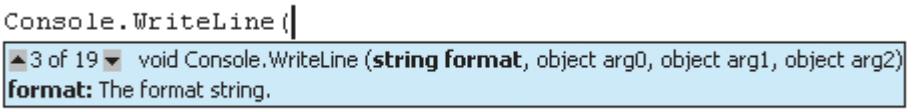
Slika 6-4:
Korištenje preopterećenih metoda pri pisanju kôda – Visual Studio će vam ponuditi popis svih preopterećenih metoda napisanog imena, s tipovima njihovih parametara, a kroz njih se možete kretati korištenjem kursorskih strelica prema gore i dolje.



Kao što uočavate, radi se o metodi za zbrajanje. Ukoliko joj se proslijede brojevi, ona će vratiti njihov zbroj. No ukoliko joj se proslijede znakovni nizovi, primjerice "Dobar" i "dan!", ona će ih konkatenirati i između njih umetnuti razmak, što bi rezultiralo s "Dobar dan!". Ovo je sasvim jednostavan primjer, no dovoljan za shvaćanje koncepta.

Mnoge često korištene metode iz .NET-a su preopterećene. Primjerice, pišete li aplikacije za konzolu, tj. bez grafičkog sučelja, često se susrećete s *Console.WriteLine* metodom koja ispisuje liniju teksta. Ona pak ima čak 19 preopterećenih metoda, koje se pozivaju ovisno o proslijeđenim parametrima (posebna metoda se poziva ukoliko želite ispisati broj, posebna ukoliko želite ispisati tekst itd.). Na slici 6-5 prikazana je pomoć pri pisanju te parametri za jednu od njih.

Slika 6-5:
Popis preopterećenih metoda za Console.WriteLine

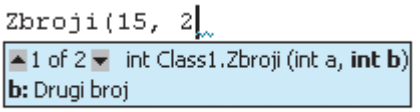


Izgubljeni u metodama

Razvojna okolina Visual Studia predstavlja idealnu pomoć u pisanju kôda. Zahvaljujući IntelliSense tehnologiji, pri pisanju metoda nudi vam se pomoć u obliku opisivanja ulaznih parametara, opisa funkcionalnosti metoda i slično. Vi i za svoje metode možete pisati pomoć koja će se pojaviti pri pisanju njihovih poziva. Pozicionirajte se u redak iznad njihove definicije i napišite "///" (tri *slasha*) i pojavit će se mali XML kôd u koji možete upisati opis metode, opis svih parametara koje prima te opis rezultata koji vraća.

```
/// <summary>
/// Opis metode.
/// </summary>
/// <param name="a">Opis prvog
parametra.</param>
/// <param name="b">Opis drugog
parametra.</param>
/// <returns>Opis rezultata
metode.</returns>
```

Tako upisani tekst poslužit će vam u trenutku kad trebate opisanu metodu pozvati iz kôda. Naime, kao na slici 6-6, pojavit će se opis svih parametara koji će se mijenjati kako budete prelazili na drugi parametar. Isto tako, primaknete li miša već gotovom pozivu u kodu, pojavit će vam se *tooltip* koji opisuje njenu funkcionalnost. Želite li pisati uredan kôd, koji mogu i drugi koristiti, svakako pišite komentare uz svoje metode – samo iznad njihove definicije napišite spomenuta tri *slasha* ("///") i pojavit će se predložak koji trebate ispuniti svojim komentarima.



Slika 6-6:
Pomoć pri pisanju poziva metoda izvući će se iz vaših komentara uz svaku metodu.

Preopterećenje operatora

U nekim specifičnim situacijama vrlo vam dobro može doći i mogućnost preopterećenja operatora. Možda poželite za neki tip podataka definirati drugačije ponašanje za standardne operatore, kao što su zbrajanje, množenje, usporedbe (veće, manje, jednako) ili logičke operacije. Uzmimo za primjer da radite s jednostavnom klasom koja prati ime i prezime te plaću svakog zaposlenika.

```
public class Zaposlenik
{
```

```
    string ImePrezime;
    int Placa;
}
```

Što ako želite na jednostavan način zbrojiti njihove plaće? To možete učiniti vrlo jednostavno – trebate samo preopteretiti operator zbrajanja. No prvo pogledajmo kako izgleda sintaksa preopterećenja operatora:

```
static public tip_podataka operator op (Parametar1 [, Parametar2])
{
    implementacija
}
```

Italik stilom označeni su dijelovi koje trebate sami promijeniti. Pod *tip_podataka* treba staviti tip koji vraća metoda. *Op* je operator, primjerice +, -, >, <, != itd. Primijetite da preopterećeni operator mora biti definiran kao *static*, što je i logično, jer je to zajednički operator za sve objekte tog tipa.

Svaki preopterećeni operator mora biti definiran kao *public* i *static*.

Dakle, preopterećeni operator mora biti definiran unutar same klase. Evo kako bi izgledao operator koji bi pri zbrajanju dvaju zaposlenika vratio zbroj njihovih plaća:

```
public class Zaposlenik
{
  public string ImePrezime;
  public int Placa;

  static public int operator + (Zaposlenik a, Zaposlenik b)
  {
          return a.Placa + b.Placa;
  }
}
```

Preopterećeni operator "+" vraća cjelobrojnu vrijednost, a kao parametre prima dva objekta, tj. dva zaposlenika. Njegova funkcionalnost je jednostavna – samo izračunava i vraća zbroj plaća dvaju

zaposlenika. U samom kodu više se ne trebate brinuti o zbrajanju – ukoliko kao parametre pošaljete objekte zaposlenika, tu će funkcionalnost obaviti netom definirani operator.

```
Zaposlenik a = new Zaposlenik();

a.ImePrezime = "Marko Marić";
a.Placa = 5700;

Zaposlenik b = new Zaposlenik();
b.ImePrezime = "Pero Perić";
b.Placa = 7100;

Console.WriteLine(a + b); // Ispisat će "12800"
```

Nasljeđivanje

Nasljeđivanje (*inheritance*) je koncept koji omogućava preuzimanje metoda, varijabli i ostalih članova od drugih klasa. Zahvaljujući nasljeđivanju, imate mogućnost stvaranja klasa koje implementiraju jednostavnu i svima zajedničku funkcionalnost, a zatim stvaranja novih specifičnih klasa koje nasljeđuju sve mogućnosti od tih jednostavnih, te ih nadograđuju novom funkcionalnošću.

Vratimo se na naš primjer s automobilima. Recimo da imate definiranu klasu *Automobil* koja u sebi ima definirane metode za pokretanje, zaustavljanje, skretanje te varijablu u kojoj je spremljena količina benzina u spremniku. Želite li pak napraviti klasu *Kamion*, i u njoj ćete morati definirati metode za pokretanje, zaustavljanje i skretanje te varijablu za količinu spremnika u benzinu. Mnogo je jednostavnije to izvesti nasljeđivanjem – klasa *Kamion* sve će članove naslijediti od klase *Automobil*, a dodat će i neke svoje nove članove.

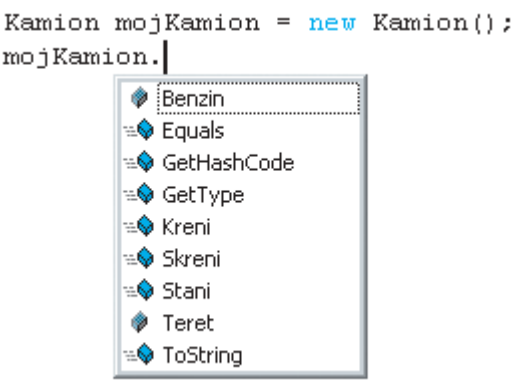
Pogledajmo prvo kako izgleda klasa *Automobil*:

```
class Automobil
{
    public int Benzin;

    public void Kreni()
    {
        // implementacija
    }

    public void Stani()
    {
```

```
        // implementacija
    }

    public void Skreni(int smjer)
    {
        // implementacija
    }
}
```

Naša nova klasa *Kamion* ima vrlo sličnu funkcionalnost kao i *Automobil*, no uz dodatak jednog novog člana, varijable zadužene za praćenje tereta u kamionu. Stoga će ona naslijediti klasu *Automobil*. Nasljeđivanje se vrši tako da se uz ime nove klase stavi dvotočka (":") i navede ime klase od koje se nasljeđuje.

```
class Kamion : Automobil
{
    public int Teret;
}
```



Slika 6-7:
Uočite da će objekt klase Kamion imati na raspolaganju i sve javne metode klase Automobil od koje nasljeđuje.

Klasa može nasljeđivati samo od jedne nad-klase (koja se u tom slučaju naziva baznom klasom), tj. u našem primjeru *Kamion* može nasljeđivati samo klasu *Automobil*. U prijevodu, neka klasa ne može svoju funkcionalnost naslijediti od više klasa, primjerice od klase *Automobil* i od klase *MotornoVozilo*.

Iako vam se na prvi pogled može učiniti da klasa *Kamion* ima samo jedan član, to nije istina – ona je naslijedila svu funkcionalnost klase *Automobil*, baš kao što je vidljivo i na slici 6-7. Korištenjem nasljeđivanja vrlo lako možete nadograđivati funkcionalnost klasa.

S druge strane, jedna klasa može istovremeno nasljeđivati od više sučelja, jer sučelja ne implementiraju funkcionalnost, već to ostavljaju samim klasama, no to će biti objašnjeno kasnije u poglavlju.

No tu cijela priča ne prestaje. Klasu *Kamion* može naslijediti neka nova klasa *KamionPrikolica* koja će naslijediti svu njegovu funkcionalnost (dakle, uključujući i funkcionalnost klase *Automobil*) te dodati neku novu.

Glavni objekt

Svi objekti dostupni u .NET Frameworku i nastali iz vaših klasa nasljeđuju u od glavnog objekta, *System.Object* bazne klase. Tako ste vjerojatno na slici 6-7 uočili neke članove koji ne pripadaju vašoj klasi, što znači da ih je ona od nekog naslijedila. *System.Object* definira nekoliko metoda – *Equals* uspoređuje jednakost između dvije instance odnosno radi li se o istom objektu. Vrlo je važna i već spominjana *ToString* metoda koja će ispisati tekst koji opisuje objekt. Primjerice, ispišete li *mojKamion.ToString()* za rezultat ćete dobiti *namespace* klase i njeno ime (primjerice, *mojeKlase.Kamion*). Ukoliko pak *ToString()* metodu pozovete nad nekim cijelim brojem, dobit ćete tekst koji sadrži njegovu vrijednost. Njega pak možete proslijediti nekoj metodi koja prima samo tekstovne parametre.

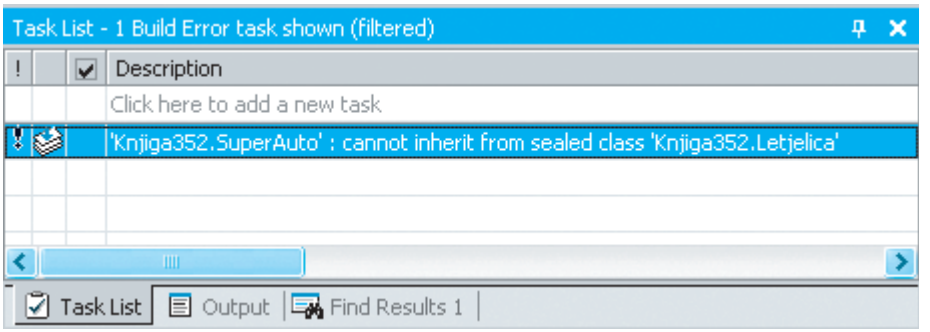
Iako po *defaultu* svaku klasu možete naslijediti u nekoj drugoj klasi, možete definirati i klase koje ne dozvoljavaju nasljeđivanje. Primjerice, možete u svojem programu koji se bavi automobilima definirati klasu *Letjelica* i onemogućiti njeno nasljeđivanje, jer ionako ne postoji niti jedna druga klasa (vozilo) koja bi mogla iskoristiti njene mogućnosti. Tada ćete napisati:

```
sealed class Letjelica
{
   // implementacija
}
```

Među klasama u .NET biblioteci ima dosta *sealed* klasa, tj. onih koje ne dozvoljavaju nasljeđivanje. Stoga se nemojte čuditi ukoliko poželite nadograditi njihovu funkcionalnost, a kompajler

vam ne dozvoli. U kasnijim poglavljima ove knjige prijeći ćemo na naprednije teme i pokazivati kako nadograđivati funkcionalnost klasa .NET Frameworka, te je zato važno znati prepoznati klase koje to ne dozvoljavaju.

Slika 6-8:
Pokušate li naslijediti od klase koja to ne dozvoljava, kompajler će vam javiti grešku.



Prekoračenje članova klasa

Dakle, već smo naučili da klasa koja nasljeđuje neku drugu klasu preuzima svu njenu funkcionalnost. No uz puko preuzimanje njenih članova, nova klasa može svakog od njih prekoračiti (engl. *override*) i dati mu novu implementaciju. Tako možete zamijeniti metode i izraditi ih na drugi način.

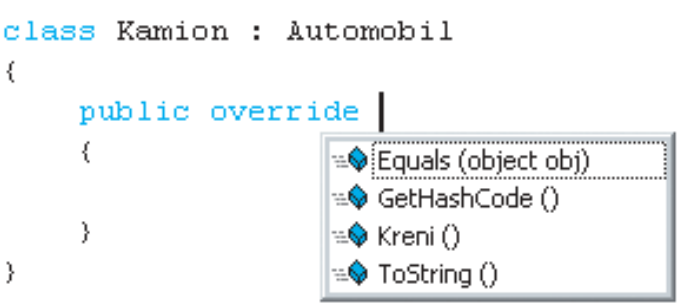
Konkretno, klasa *Automobil* ima metodu *Kreni,* koja pokreće automobil. No klasa *Kamion* može definirati drugačije ponašanje za metodu *Kreni*. Iako ju je ona naslijedila, može je redefinirati i pridijeliti joj drukčiju funkcionalnost.

Prvo i najvažnije – u definiciji klase *Automobil* morate naznačiti da sve klase koje ju nasljeđuju mogu redefinirati odnosno prekoračiti metodu *Kreni*. To postižete ključnom riječju *virtual*.

```
class Automobil
{
    public virtual void Kreni()
    {
        // implementacija
    }

    // ...
}
```

Sada u klasi *Kamion* samo trebate definirati novu metodu istog imena s istim parametrima i označiti je ključnom riječi *override*.

```
class Kamion : Automobil
{
   public override void Kreni()
   {
      // nova implementacija
   }

   // ...
}
```



Slika 6-9:
Čim pri definiranju nove klase napišete ključnu riječ override, Visual Studio će vam ponuditi popis klasa koje možete prekoračiti.

Oprez, oprez, slijedi važna napomena! Iako bi se moglo zaključiti da metode koje nisu označene s *virtual* ne možete prekoračiti u novim klasama, to nije u potpunosti istina. Naime, mnoge metode iz baznih .NET klasa neće biti *virtual*, no vi ih svejedno možete prekoračiti tako da pri njihovoj deklaraciji u novim klasama navedete ključnu riječ *new*, primjerice:

```
internal new void Kreni() {  // itd.
```

Tako zapravo stvarate novu implementaciju neke metode i potpuno ste neovisni o originalu iz bazne klase.
No samo metode koje su definirane s ključnom riječi *virtual* moći će se ponašati po konceptima višeobličja, što je objašnjeno kasnije u poglavlju. Ukoliko neku nevirtualnu metodu prekoračite ključnom riječi *new*, za nju neće vrijediti pravila višeobličja.

Ponekad ćete ipak poželjeti pozvati funkcionalnost bazne klase. Primjerice, ukoliko se metode *Kreni* u klasi *Automobil* i klasi *Kamion* razlikuju samo u tome što se u klasi *Kamion* provjerava je li teret ukrcan, jednostavnije će biti u prekoračenoj klasi dodati tu provjeru, a ostatak prepustiti baznoj klasi, tj. onoj u metodi *Automobil* koja je zadužena za paljenje svjetala, pokretanje vozila, prebacivanje u prvu brzinu itd.

Baznoj klasi se pristupa korištenjem ključne riječi *base*. Evo kako bi izgledala metoda *Kreni* u klasi *Kamion*:

```
class Kamion : Automobil
{
   public override void Kreni()
   {
      // provjeri teret

      base.Kreni();
   }

   // ...
}
```

Dakle, korištenjem ključne riječi *base* pozvala se metoda *Kreni* u baznoj klasi, tj. klasi *Automobil*. U njoj je definirana ostala funkcionalnost pokretanja vozila, a u klasi *Kamion* samo smo još dodali provjeru tereta (unutar komentara).

Tek kad razumijete nasljeđivanje možete u potpunosti razumjeti sve ključne riječi koje određuju prava pristupa metodama. Konkretno, možda je ostalo nejasno kakve su to *protected* klase. Primjerice, da ste u klasi *Automobil* metodu *Stani* definirali kao *private*, nju ne biste naslijedili u klasi *Kamion*, jer je ona, po definiciji, dostupna samo unutar klase u kojoj je definirana (dakle *Automobil*). No da ste je definirali kao *protected*, mogli biste joj pristupiti iz klase *Kamion*. Ipak i dalje to ne biste mogli iz glavnog programa, jer, po definiciji, *protected* članovima je dozvoljen pristup samo iz klasa u kojima su definirani te iz klasa koje ih nasljeđuju.

Sučelja

Već prije smo objasnili što su *sučelja* (engl. *interfaces*) u OOP-u, no nije naodmet ponoviti. Dakle, radi se o točno definiranim članovima koji služe za komuniciranje sa samim objektom. Sučelje

samo definira članove koji će biti dostupni njegovom implementacijom, a samu implementaciju prepušta klasama koje ga nasljeđuju. Tako možete s različitim objektima koji implementiraju isto sučelje komunicirati na isti način.

Objasnimo sve na najpoznatijem primjeru iz literature. Dakle, pretpostavimo da postoji sučelje imena *IOblik* koje u sebi sadrži metodu za računanje površine. To sučelje mogu implementirati različite klase, primjerice klasa *Kvadrat* i klasa *Krug*. To u praksi znači da obje klase moraju sadržavati metodu za računanje površine, no svaka će je klasa izračunati na svoj način.

Iako biste taj primjer na prvi pogled, primjenjujući netom naučeno, mogli izvesti korištenjem klasa, tj. tako da postoji klasa *Oblik* u kojoj je definirana metoda za računanje površine te da nju nasljeđuju klase *Kvadrat* i *Krug* i prekoračuju metodu za računanje površine, to ipak nije ispravno. Prvi razlog je u tome što klasa *Oblik* mora imati implementiranu metodu za računanje površine, što ne možemo očekivati, jer ne znamo o kakvom se obliku radi. Drugi razlog je što se nasljeđivanjem od klase *Oblik* novim klasama nikako ne može nametnuti obaveza da implementiraju svoju metodu za računanje površine, jer mogu koristiti i onu od klase *Oblik*, koju pak ne znamo implementirati bez informacije o kakvom se obliku radi. Sučelja rješavaju oba problema – ona samo definiraju metode koje klase koje ih nasljeđuju moraju imati, a pritom ih ne implementiraju (što nam treba za metodu za računanje površine) te svim klasama koje ih nasljeđuju nameću obavezu implementacije svih metoda definiranih u sučelju.

Definiranje sučelja

Sučelje se definira ključnom riječju *interface*. Za razliku od klasa, svi članovi moraju biti definirani bez ključne riječi koja određuje prava pristupa, poput *public*, *private* itd. Prava pristupa određuje samo ključna riječ navedena uz definiciju sučelja – napišete li *public interface*, svi će njegovi članovi biti *public*.

```
public interface IVozilo
{
    void Kreni();
    void Stani();
    void Skreni(int smjer);
}
```

Kao što vidite, definirano je sučelje *IVozilo*, no bez implementacije samih metoda. Sve su metode definirane svojim tipom podataka koji vraćaju (u našem slučaju, niti jedna ne vraća ništa i sve su *void*) te parametrima koje prima.

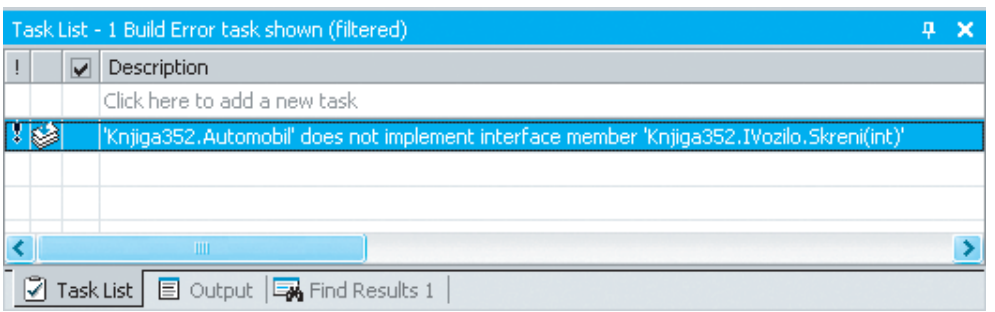
Klase se obavezuju implementirati sučelje tako da ga jednostavno naslijede. Sintaksa je ista:

```
class Automobil : IVozilo
{
```

```
    public void Kreni() {
        // implementacija
    }

    // obavezno implementirati sve metode iz sučelja IVozilo
}
```

Sve što trebate napraviti da bi klasa bila ispravna jest implementirati sve metode definirane u sučelju koje je naslijeđeno. Ukoliko to ne učinite, tj. ne napišete metode *Kreni*, *Stani* i *Skreni* istih parametara i povratnih tipova varijabli, kompajler će vam javiti grešku.

Slika 6-10:
Kompajler je javio grešku jer nismo implementirali metodu Skreni.



Primijetite da pri implementiranju sučelja ne trebate koristiti ključnu riječ *override*, jer ne prekoračujete postojeće metode. Vi ih samo implementirate i dajete im funkcionalnost, pa sve metode sučelja definirate na standardan način, baš kao što biste činili i da se ne nasljeđuje od sučelja.

No, za razliku od nasljeđivanja klasa, pri čemu možete naslijediti samo jednu klasu, sučelja su mnogo slobodnija – dozvoljeno vam je naslijediti koliko god sučelja želite. Naravno, tada morate implementirati svako od njih. Sučelja koja se nasljeđuju moraju se odvojiti zarezom pri definiciji.

```
class Kamion : IVozilo, ITeretnjak
{
```

```
    // obavezno implementirati sve metode iz
    // sučelja IVozilo i ITeretnjak
}
```

Ne možete stvoriti objekt iz sučelja. Sučelje u sebi, za razliku od klasa, ne sadrži implementaciju, pa ako pokušate instancirati novi objekt iz nekog sučelja (primjerice "IVozilo mojeVozilo = new IVozilo()"), dobit ćete informaciju o greški "Cannot create an instance of the abstract class or interface".

Implementacija varijabli sučelja

Osim metoda, kao u prethodnim primjerima, sučelja mogu definirati i varijable koje svaka klasa mora implementirati. U tom je slučaju postupak malo složeniji nego pri jednostavnom deklariranju nove varijable. U sučelju možete definirati hoće li neka varijabla biti samo za čitanje, samo za pisanje ili oboje. Evo i kako:

```
public interface IVozilo
{
    int Benzin
    {
        get;
        set;
    }
}
```

Kao što vidite, u sučelju smo definirali *integer* varijablu *Benzin* koja je i za čitanje (tome služi *get*) i za pisanje (tome služi *set*). Ukoliko biste željeli definirati da se varijabla *Benzin* u klasama koje nasljeđuju sučelje ne može mijenjati iz ostatka programa, maknuli biste *set*, a ukoliko želite spriječiti njeno čitanje, maknut ćete *get*.

No tu nije kraj. Sad morate u svakoj klasi koja naslijedi to sučelje implementirati metode *get* i *set* za postavljanje vrijednosti varijabli. To ipak nije složeno kao što se čini.

```
class Automobil : IVozilo
{
    private int b;
```

```
    public int Benzin
    {
        get
        {
            return b;
        }
        set
        {
            b = value;
        }
    }
}
```

Ključna je varijabla *b* koja je označena s *private* pravom pristupa. U nju će se interno spremati vrijednost stanja benzina. To se omogućava uz pomoć *get* i *set* metoda definiranih unutar vitičastih zagrada varijable *Benzin*. *Get* metoda je jednostavna i njen je zadatak pročitati varijablu *b* u kojoj je interno spremljena vrijednost benzina i vratiti je. Metoda *set* pak postavlja varijablu *b* na vrijednost *value* – radi se o vrijednosti koja se iz vanjskog programa želi upisati u varijablu *Benzin*.

Evo i primjera:

```
Automobil mojAuto = new Automobil();
mojAuto.Benzin = 15;
Console.WriteLine(mojAuto.Benzin);
```

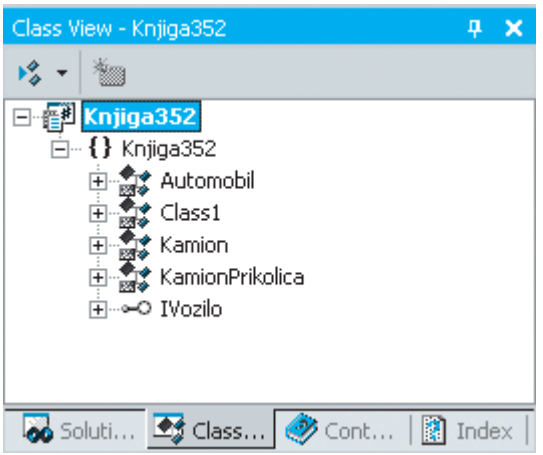
Prvo se instancira objekt tipa *Automobil*. Zatim se upisuje vrijednost *Benzin* varijable. Iza scene se zapravo u tom trenutku poziva *set* metoda, koja u varijablu *b* upisuje proslijeđenu vrijednost odnosno 15. Uočite da je varijabla *b* označena s *private*, tj. ne može joj se pristupiti iz vanjskog programa i cijela komunikacija s njom ide preko varijable *Benzin*.

Dok ispisujemo vrijednost varijable *Benzin* poziva se njena *get* metoda koja vraća vrijednost varijable *b*. Naravno, vi možete cijelu stvar malo zakomplicirati te dodati još poneku funkcionalnost u *get* i *set* metode, no ovo će u većini slučajeva biti dosta.

Višeobličje

Evo nas napokon i na pravim primjerima objektno orijentiranog programiranja. Ono što slijedi najbolji je pokazatelj naprednih mogućnosti objektno orijentiranih jezika. Dakle, radi se o već prije objašnjenom konceptu, a sad ćemo ga iskušati i u praksi.

Već smo spominjali da koristeći metode nekog sučelja možemo komunicirati sa svim klasama koje ga implementiraju. Sljedeći primjer to najbolje pokazuje. Naime, napravit ćemo metodu koja će



Slika 6-11:
Class View omogućava vam pregled svih klasa i sučelja u vašem programu.

Pogled svisoka

Osim ručnog pisanja kôda i stvaranja klasa i njihovih članova, možete se poslužiti i *Class Viewom* smještenim u desnom stupcu Visual Studia (ukoliko ga ne vidite, odaberite opciju *View – Class View* ili pritisnite tipkovničku kraticu Ctrl + Shift + C). Primjerice, da biste dodali novu varijablu u sučelje, kliknite na sučelje u *Class Viewu* desnim gumbom miša i odaberite opciju *Add – Add Property*. Pojavit će vam se prozor kao na slici 6-12, a u njemu možete definirati novu varijablu te odabrati je li samo za čitanje (*get*), za pisanje (*set*) ili oboje (*get/set*).

Naravno, *Class View* omogućava vam i druge korisne stvari. Želite li stvoriti novu klasu, kliknite desnim gumbom miša na najviši element u popisu koji odražava ime vašeg programa te odaberite opciju *Add – Add Class*. Otvorit će vam se prozor u kojem možete definirati sve postavke nove klase: prava pristupa, tip, odrediti koju klasu ili sučelje nasljeđuje itd.

Poigrajte se s *Class Viewom* jer skriva dosta korisnih opcija, koje mogu uvelike ubrzati stvaranje novih klasa i njihovih članova. Postupak je jednostavan – kliknite desnim gumbom miša na bilo koji element i proučite što vam se sve nudi.

Slika 6-12:
Automatsko dodavanje varijable u sučelje

iskoristiti funkcionalnost nekog sučelja, a potpuno je svejedno objekt koje klase joj ga proslijedimo, sve dok te klase implementiraju to sučelje.

Za početak pretpostavimo da imamo dvije klase, *Automobil* i *Kamion*, a obje implementiraju sučelje *IVozilo*. Pogledajmo metodu koja može komunicirati s obje klase:

```
void Voznja(IVozilo vozilo)
{
  if (vozilo.Benzin > 10)
  {
            vozilo.Kreni();
            vozilo.Skreni(1);
            vozilo.Stani();
  }
}
```

Radi se o metodi koja simulira neku vožnju. Pogledate li njenu implementaciju, uočit ćete da koristi sve metode sučelja *IVozilo*. Njoj možemo proslijediti objekt tipa *Automobil* i objekt tipa *Kamion*, jer oba implementiraju sučelje *Ivozilo,* te stoga neće biti problema – vozilom će se normalno upravljati odnosno pozivati sve metode.

```
Automobil mojAuto = new Automobil();
Kamion mojKamion = new Kamion();

Voznja(mojAuto);
Voznja(mojKamion);
```

Objekti proslijeđeni metodi *Voznja* automatski će biti pretvoreni (*cast*) u objekt tipa *IVozilo*. Kako sučelje nameće obavezu implementacije svih njegovih članova, možete biti sigurni da sve metode i varijable korištene u metodi *Voznja* postoje te da će ona biti ispravno izvršena.

Dakle, to je pravo *višeobličje* – s različitim objektima koji implementiraju isto sučelje može se komunicirati na isti način koristeći metode tog sučelja.

Iako je prije bilo spomenuto da ne možete instancirati objekt iz sučelja (koristeći *new* operator), možete ipak deklarirati objekt tog tipa. Pogledajte primjer:

```
Automobil mojAuto = new Automobil();
IVozilo vozilo;

vozilo = (IVozilo) mojAuto;
vozilo.Kreni();
```

Apstraktne klase

Iako vam se može učiniti da su klase i sučelja sve što vam ikad može zatrebati pri programiranju, to nije sve! Po funkcionalnosti negdje između klasa i sučelja nalaze se *apstraktne klase*.

Možda ćete ponekad željeti stvoriti klasu koja implementira neke metode, no isto tako ostavlja implementaciju drugih metoda klasama koje ju nasljeđuju . Kao što vidite, radi se o kombinaciji običnih klasa (zato jer i same implementiraju neku funkcionalnost) i sučelja (jer prepuštaju implementaciju druge funkcionalnosti klasama koje ih nasljeđuju). Rezultat toga su, dakle, apstraktne klase, koje – poput sučelja – moraju biti naslijeđene i ne mogu se koristiti same za sebe (baš zbog tih neimplementiranih, ali definiranih metoda).

```
public abstract class
MojaApstraktnaKlasa
{
   public abstract void
ApstraktnaMetoda();
   public void
FunkcionalnaMetoda() {
      // implementacija
   }
}
```

Primijetite ključnu riječ *abstract* – njome se određuje apstraktna klasa. Ona također služi i za definiranje apstraktnih metoda. U primjeru je definirana *ApstraktnaMetoda* koju moraju implementirati sve klase koje nasljeđuju apstraktnu klasu (kao i kod svih metoda u sučeljima). No možete i definirati neku metodu s implementacijom (u našem primjeru *FunkcionalnaMetoda*) – njih možete označiti i s *virtual*, čime dozvoljavate da sve klase koje nasljeđuju apstraktnu klasu implementiraju svoju funkcionalnost tih metoda.

Pri nasljeđivanju i implementaciji apstraktne klase sve apstraktne metode morate prekoračiti korištenjem ključne riječi *override*. Tu se one razlikuju od sučelja, čije članove ne trebate prekoračiti korištenjem *override*, već samo implementirati na standardan način. Dakle, želite li naslijediti prethodnu klasu, evo kôda:

```
public class NovaKlasa :
MojaApstraktnaKlasa
{
   public override void
ApstraktnaMetoda()
   {
      // implementacija apstrakt-
ne metode
   }
}
```

Primijetite da niste trebali naslijediti i implementirati metodu *FunkcionalnaMetoda*, jer njena implementacija već postoji u apstraktnoj klasi i jednostavno je od nje naslijeđena.

Objekt *mojAuto* eksplicitno je pretvoren (*cast*) u objekt tipa *IVozilo* (naravno, uvjet za ovo je da klasa *Automobil* implementira sučelje *IVozilo*). Objektu tipa *vozilo* dostupne su sve metode sučelja *IVozilo*, što je i očekivano, no uočavate li potencijalan problem? U sučelju *IVozilo* ne postoje njihove implementacije, ali to ne smeta – koristit će se od one klase iz koje je objekt pretvoren u objekt tipa *IVozilo*. Konkretno, koristit će se metode klase *Automobil*, jer je objekt *vozilo* prije pretvaranja bio tipa *Automobil*.

Slične mogućnosti vam se pružaju i pri korištenju klasa. Vratimo se na stariji primjer u kojem imamo klasu *Automobil* te klasu *Kamion* koji je nasljeđuje i nadopunjava novom funkcionalnošću. Dakle, klasa *Automobil* je bazna klasa. Sve klase koje ju nasljeđuju mogu se, zahvaljujući višeobličju, ponašati kao ona.

Tako možete stvoriti sličnu metodu *Voznja,* koja će za parametar primati objekt tipa *Automobil*. Ukoliko je pak pozovete proslijedivši joj objekt tipa *Kamion*, on će biti automatski pretvoren u svoju baznu klasu. Naravno, sve dodatne metode koje je implementirala klasa *Kamion* postat će nedostupne, a vidljivi će biti samo članovi bazne klase.

Evo nas i na kraju poglavlja koje se bavi objektno orijentiranim programiranjem. Ono je bilo posve nužno za razumijevanje rada .NET-a jer, kao što ćete vidjeti u kasnijim poglavljima, gotovo svaki dio .NET-a, svaka njegova klasa i sve vaše želje za nadograđivanjem njihovih mogućnosti, rezultirat će potrebom za korištenjem koncepata OOP-a.

Primjerice, želite li napraviti vlastito tekstualno polje za Windows aplikaciju koje će, upišete li u njega tekst "tajna" promijeniti boju, trebate samo napraviti novu klasu koja nasljeđuje klasu za tekstualno polje i dodati joj tu sitnu funkcionalnost. To je, naravno, samo vrh ledenog brijega, a vašoj mašti je prepušteno istraživanje drugih mogućnosti.

7. POGLAVLJE

Iznimke

U ovom poglavlju:

- Pogreške u kódu
- Mehanizmi hvatanja iznimki
- Hijerarhija iznimki u .NET-u
- Pravilno hvatanje iznimki

Nemojte se zavaravati – pogreške u kódu su veoma stvarne i ne događaju se nekom drugom! Pritom uopće nije bitno u kojem programskom jeziku radite ili koji razvojni alat koristite, jer su programske greške vezane uz koncept i stil programiranja. Isto tako, greške se vrlo često događaju i zbog različitih vanjskih uvjeta, bez znanja programera (primjerice, pokušavate pristupiti bazi podataka koja je baš u tom trenutku nedostupna zbog administriranja), pa skoro da i nije moguće da vaš kôd nema niti jednu grešku, već je samo bitno da ih na pravi način uočavate i predviđate mjesta na kojima bi se mogle pojaviti.

Cilj ovog poglavlja je pokazati vam kako se, pišući kôd u .NET-u, možete nositi s greškama i iznimnim situacijama. Tako ćete naučiti kako iskoristiti najvažnije mogućnosti u svrhu pisanja ispravnog kôda te kako u razdoblju pisanja i testiranja svojih aplikacija pronaći greške.

.NET ima još jednu posebnost o kojoj je već bilo riječi, a zove se *Garbage Collection*. Radi se o metodi upravljanja memorijom koja u potpunosti otklanja mogućnost grešaka

programera i zauzimanja memorije koja se više ne koristi, što bi dovelo do nepotrebnog zauzimanja resursa računala.

Iznimke

Krenimo redom i upoznajmo se najprije s iznimkama i metodama hvatanja iznimaka. Vrlo je važno da razumijete što su to iznimke. Ukratko rečeno, radi se o situacijama u kojima vaš kôd radi neispravno. Primjera je doista bezbroj: iznimke će se tako desiti kad pokušate pisati u datoteku koja ima postavljen atribut *Read Only*, kad pokušate napraviti matematičku operaciju nad dva znakovna niza (što je moguće samo nad brojevima) ili pak kad u vašem programu pokušate pristupiti nekom resursu na Internetu, a ne postoji aktivna veza na Internet. Naravno, tipična pogreška koja uzrokuje iznimku je i pokušaj dijeljenja s nulom, što bi rezultiralo beskonačnim brojem koji računalo nije u stanju prikazati.

Sve te situacije rezultirat će prekidanjem rada vašeg programa i ispisivanjem greške. Rješenje za te probleme je jednostavno – trebate samo ostvariti mehanizme hvatanja iznimki i upravljanja njima. Primjerice, u slučaju greške pri pokušaju pisanja u *Read Only* datoteku, korisniku trebate ponuditi spremanje datoteke pod nekim drugim imenom, a pri dohvaćanju nekog resursa s Interneta kad ne postoji aktivna veza samo trebate ispisati odgovarajuću poruku u kojoj ćete upozoriti korisnika da se treba spojiti na Internet.

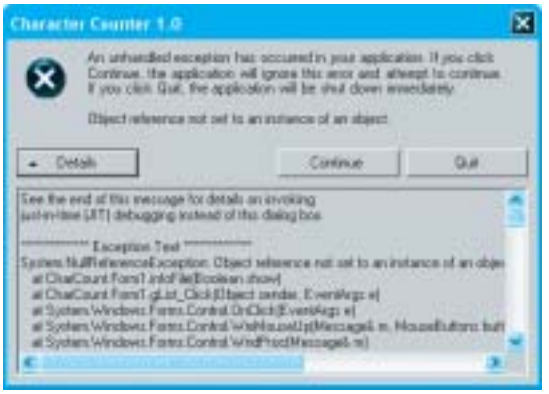
Korištenjem .NET-a imate idealne mogućnosti hvatanja iznimki. Tako možete sav kôd koji će baratati greškama držati na odvojenom mjestu te tako imati čist i pregledan glavni kôd. Naravno, hvatanjem iznimki možete vrlo lako uočiti probleme koji se često ponavljaju u vašem kodu u periodu testiranja te ih lako ukloniti.

Mehanizmi hvatanja iznimki

U C#-u vam na raspolaganju stoji nekoliko naredbi koje zajedno čine strukturu za hvatanje iznimki. Radi se o bloku *try catch finally* – kasnije ćemo detaljno objasniti svaku od naredbi, a prvo ćemo objasniti cijeli blok i njegovu namjenu.

Kamo bježe neuhvaćene iznimke?

Sve iznimke koje ne uhvatite u svojim aplikacijama i na njih pravilno ne reagirate izazvat će grešku u radu aplikacije, bilo da se radi o web-aplikaciji u ASP.NET-u ili o desktop aplikaciji pisanoj uz pomoć Windows Formsa. Rezultat jedne pogreške možete vidjeti na slici 7-1, a radi se o nečemu što ne želite da vide vaši korisnici.



Slika 7-1: ***Greška u Windows Forms aplikaciji***

```
try {

   // Ovdje se nalazi "problematični" kôd koji bi mogao
   // uzrokovati pojavljivanje greške.

}
catch (InvalidCastException) {

   // Kôd koji se oporavlja od InvalidCastException iznimke
   // i svih drugih proizašlih iz nje
   // (detaljnije o "tim drugim" iznimkama kasnije).

}
catch (Exception e) {

   // Kôd koji hvata sve iznimke
   // (zato jer sve iznimke proizlaze iz Exception iznimke).

}
finally {

   // Kôd koji "čisti" sve operacije započete u try bloku.
   // Taj se kôd uvijek izvršava, neovisno o pojavljivanju
   // iznimke.

}
```

U C#-u se mogu pojaviti samo CLS (*Common Language Specification*) iznimke, a to su one proizašle iz System.Exception (kasnije ćemo detaljnije objasniti hijerarhiju iznimki, no sad je najvažnije da shvatite da se na vrhu hijerarhije nalazi System.Exception, a sve druge su ispod nje, pa kažemo da su *proizašle* iz nje). Imate li *catch* blok koji hvata iznimku tipa *Exception*, zapravo hvatate sve moguće iznimke.

Ovdje postoje dvije *catch* naredbe – vi ih možete imati koliko god želite i svakom možete hvatati posebnu vrstu iznimki, no najčešće će to biti samo jedna ili dvije *catch* naredbe.

try blok

U *try* bloku se nalazi "problematičan" kôd koji bi mogao izbaciti grešku i uzrokovati pojavljivanje iznimke. Takvih primjera ima jako mnogo, neke smo spomenuli već na početku teksta, a evo ih još nekoliko: možda pokušavate pristupiti bazi podataka za koju nemate određene dozvole ili pak želite obraditi neke podatke koje je korisnik upisao, no oni su upisani u krivom obliku.

Važno je primijetiti da ste vi kao programer uočili potencijalne probleme i taj kôd smjestili unutar *try* bloka, što vam omogućava hvatanje i pravilno rukovanje s pogreškama.

catch blok

U *catch* bloku se nalazi kôd koji će se izvršiti u slučaju pojavljivanja iznimke. Izraz u zagradi nakon *catch* naredbe je zapravo filtar i on određuje u slučaju koje iznimke će se izvršiti kôd. U prethodnom primjeru imali smo *InvalidCastException* filtar koji bi uhvatio sve iznimke *InvalidCastException* tipa i sve one koje su, po hijerarhiji, proizašle iz nje.

Obavezno catch blokove posložite tako da specifične iznimke idu na vrhu, a općenite na dnu. Ukoliko pak napravite obrnuto, čak će vas i kompajler upozoriti ako se više specifičnih iznimki pojavljuje pri dnu, jer se taj kôd nikad neće izvršiti.

Primijetite da se *catch* blokovi u slučaju iznimke provjeravaju od vrha prema dnu, pa je nužno staviti specifične iznimke na vrh (primjerice, *InvalidCastException*), a one općenitije pri dnu (primjerice, *Exception* koja hvata sve iznimke).

finally blok

Finally blok sadržava kôd koji će se sigurno izvršiti i taj kôd najčešće služi za čišćenje i oslobađanje resursa koji su bili korišteni u *try* bloku. Jednostavan je primjer pri korištenju datoteka:

```
using System.IO;

void koristiDatoteku(string ime)
{
   FileStream fs = null;
   try
   {
```

```
            fs = new FileStream(ime, FileMode.Open);
            // kôd koji radi s datotekom i nešto u nju zapisuje
        }
        catch (Exception)
        {
            // Pojavila se neka greška pri radu s datotekom.
        }
        finally
        {
            if (fs != null) fs.Close();
        }
    }
```

U *try* bloku pristupamo datoteci i pokušavamo nešto s njom raditi. U slučaju da se pojavi bilo kakva greška, hvatamo je *catch* blokom. No u tom slučaju datoteka ostaje otvorena i u upotrebi – zato nam je nužan *finally* blok.

Ne bi bilo ispravno staviti naredbu za zatvaranje datoteke poslije bloka za hvatanje iznimki: što ako se pojavi iznimka, no ona se ne uhvati u *catch* bloku jer se koristila neka specifična iznimka kao filtar? U tom slučaju će datoteka ostati otvorena i nakon što vaš program završi, što je neprihvatljiva situacija.

Sav kôd u njemu će se izvršiti u svakom slučaju, bilo da se pojavila greška ili je sve prošlo u redu. U slučaju da se pojavi greška, zatvorit će se datoteka, a u slučaju da je sve prošlo u redu, svejedno će se izvršiti *finally* blok i datoteka će se zatvoriti.

Ponekad vam i ne treba *finally* blok, jer jednostavno niste koristili resurse koje obavezno treba "počistiti" na kraju. Važno je samo primijetiti da *finally* blok dolazi poslije svih *catch* blokova i da može postojati samo jedan u *try catch finally* dijelu, za razliku od *catch* blokova, kojih može biti više.

Rad s iznimkama

Mehanizam upravljanja iznimkama može hvatati sve iznimke bazne klase System.Exception i svih podklasa nastalih iz nje. Stoga sve iznimke imaju neka ista svojstva koja možete koristiti pri hvatanju iznimki kao izvor korisnih informacija o grešci koja se pojavila. U tablici 7-1 nalaze se tri glavna svojstva klase *System.Exception* koje nasljeđuju sve podklase.

U praksi možete iskoristiti ova svojstva za ispisivanje informacije o grešci. No tad ćete u *catch* bloku morati, uz tip iznimke koju hvatate, upotrijebiti ime varijable koja će sadržavati sva ta svojstva. Pogledajte naredni primjer, u kojem ispisujemo informaciju o grešci (radi se o aplikaciji za konzolu):

Tablica 7-1: ***Tri glavna svojstva Exception klase***

Svojstvo	Opis
Message	Tekst pogreške koji opisuje što se desilo i zašto se pojavila iznimka
Source	Ime *assemblya* koji je stvorio iznimku
StackTrace	Ime metode, datoteka s pogreškom i linija u kojoj se pojavila pogreška

```
//...
try
{
   // kôd koji uzrokuje pogrešku
}
catch (Exception e)
{
   Console.WriteLine("Tekst pogreške: " + e.Message);
}
//...
```

Primjerice, pojavi li se greška dijeljenja s nulom, program će ispisati: "Tekst pogreške: Attempted to divide by zero".

Želite li pak ispisati detaljnu informaciju o izvoru greške, upotrijebite *StackTrace* svojstvo (u gornjem primjeru ispišite *e.StackTrace*). Dobit ćete informaciju o metodi u kojoj se pogreška pojavila, datoteci u kojoj se nalazi ta metoda i liniji na kojoj se nalazi naredba koja je uzrokovala grešku. Primjerice, ta informacija može imati sljedeći oblik, što je izrazito korisno pri otkrivanju izvora pogreške:

```
at MojaAplikacija.Class1.Main(String[] args) in
e:\code\c#\MojaAplikacija\class1.cs:line 23
```

Hijerahija iznimki

U .NET-u postoji *System.Exception* klasa koja definira općenitu iznimku, a sve druge iznimke proizlaze iz nje i realizirane su obliku njenih podklasa. Najvažnije dvije klase proizašle iz *Exception* klase su *ApplicationException* i *SystemException*. *ApplicationException* klasa vam omogućava definiranje vlastitih iznimki specifičnih za vašu aplikaciju.

SystemException su iznimke koje se mogu pojaviti bilo kad za vrijeme izvršavanja aplikacije i zato se još zovu *Runtime* iznimke (pojavljuju se u *Runtimeu* odnosno za vrijeme izvršavanja). Primjeri-

Pogreška u pogreški u pogreški u...

Što ako vam se desi pogreška u *finally* bloku? Ništa posebno – ta pogreška identična je bilo kojoj drugoj koja se pojavljuje poslije *try catch finally* bloka. No kako hvatati takve pogreške? Posve jednostavno – ako želite, možete koristiti više *try catch finally* blokova jednih u drugima. Primjerice, što ako bi se "zabunili" i u gornjem primjeru u *finally* bloku nakon zatvaranja datoteke pokušali pristupiti datoteci (primjerice, ispisati njenu duljinu uz pomoć *fs.Length*)? Kôd koji bi ispravno rukovao tom greškom može izgledati ovako:

```
try
{
   FileStream fs = null;
   try
   {
      fs = new
FileStream(ime,
FileMode.Open);
      // kôd koji radi s
datotekom i nešto u nju
zapisuje
   }
   catch (Exception)
   {
      // Pojavila se neka
greška pri radu s datotekom.
   }
   finally
   {
      if (fs != null)
fs.Close();
      // korištenje
fs.Length svojstva
   }
}
catch (Exception)
{
   // pojavila se još jedna
greška
}
```

Na raspolaganju vam stoji neograničen broj *try catch finally* blokova. Naravno, skoro uvijek će vam biti dosta samo jedan blok u kojem predvidite sve moguće pogreške, a svakako pokušajte u *catch* i *finally* blokovima pisati što jednostavniji kôd koji ne može uzrokovati dodatne pogreške i iznimke.

ce, pokušate li pristupiti nepostojećem članu nekog polja, CLR će vam dojaviti *IndexOutOfRangeException*, koja je proizašla iz *SystemException* klase. Slično, pokušate li raditi s objektom koji još ne postoji (stvoren je, no ne i instanciran), dobit ćete *NullReferenceException*.

Slijedi sažeta hijerarhija klasa s nekoliko najvažnijih tipova iznimaka, da dobijete dojam kakve sve iznimke postoje:

```
System.Exception
   System.ApplicationException
```

```
        System.Reflection.TargetException
        ...
    System.IO.IsolatedStorage.IsolatedStorageException
    System.SystemException
        System.ArgumentException
            System.ArgumentNullException
            System.ArgumentOutOfRangeException
            ...
        System.ArithmeticException
            System.DivideByZeroException
            System.OverflowException
            ...
        System.FormatException
        System.IndexOutOfRangeException
        System.InvalidCastException
        System.IO.IOException
            System.IO.DirectoryNotFoundException
            System.IO.FileNotFoundException
            ...
        System.NullReferenceException
        System.OutOfMemoryException
        System.Security.Policy.PolicyException
            System.Security.SecurityException
            System.Threading.SynchronizationLockException
            System.Threading.ThreadInterruptedException
            System.TypeLoadException
                System.DllNotFoundException
                System.EntryPointNotFoundException
            System.UnauthorizedAcessException
            ...
        ...
    ...
```

Potpunu hijerarhiju klasa možete dobiti potražite li u MSDN-u “Exception class” i odaberete li “Derived classes”.

Što nam zapravo govori prethodni popis? Pomoću njega dobivate jasniju sliku i razumijete zašto *System.Exception* hvata sve iznimke. Primjerice, pojavi li se iznimka tipa *System.DivideByZeroException*, nju biste uhvatili bilo kojim od sljedećih *catch* blokova:

```
catch (DivideByZeroException)
catch (ArithmeticException)
```

```
catch (SystemException)
catch (Exception)
```

Hijerarhija klasa iznimaka ima za posljedicu i da će *ArithmeticException* blok uhvatiti sve iznimke svojih podklasa, primjerice *System.DivideByZeroException* i *System.OverflowException*. To je najveća prednost postojanja ovakve hijerarhije. Zato *System.Exception* koji se nalazi na vrhu hijerarhije hvata baš sve.

Od mnogih iznimaka moguće je oporaviti se pravilnim hvatanjem i rukovanjem, kao što je slučaj i s, recimo, spomenutom *DivideByZeroException* iznimkom. No neke se iznimke, poput *StackOverflowException,* zbog svoje prirode (stog se napunio zbog previše poziva metoda i nema resursa za daljnje naredbe) smatraju fatalnima: skoro je nemoguće oporaviti se od njih u kôdu i gotovo sigurno će uzrokovati prekid rada aplikacije.

U C#-u postoji još jedan catch blok, koji će poput catch (Exception) hvatati sve iznimke. Jednostavnije ga je napisati, jer u sebi ne sadržava niti jedan filtar iznimki:

```
catch {
   // kôd za rukovanje s bilo kojom iznimkom
}
```

Pravilno rukovanje iznimkama

Kao i kod svih drugih tehnologija i tehnika programiranja, bez poznavanja sintakse ne možete ništa, no mnogo važnije je znati kako pojedinu tehniku pravilno upotrebljavati. Isti je slučaj i s hvatanjem iznimki. Slijedi nekoliko korisnih savjeta koje možete iskoristiti u svakoj situaciji.

Za početak, pojasnimo još jednom što podrazumijeva hvatanje iznimki. Kad napišete *try catch* blok, to znači da ste u tom kôdu očekivali iznimku, da razumijete kako se desila i da znate što s njom učiniti i kako ispravno postupiti u oporavku od greške.

Lako vas može povesti mogućnost da hvatate sve iznimke korištenjem *catch (Exception)* bloka. No što u tom slučaju želite napraviti? Zar doista mislite da se u vašem kôdu može dogoditi bilo

kakva iznimka? Zar mislite da svojim kôdom u *catch* bloku možete predvidjeti sve situacije i od njih se na pravi način oporaviti?

Hvatanje svih iznimki možda i jest korisno pri izradi aplikacija kad želite saznati detaljnije informacije o svakoj grešci koja se pojavi tako da u *catch* bloku ne napravite baš ništa, osim što ispišete informaciju o grešci. No i tad je mnogo prikladnije koristiti *debugger* koji će vam javiti što je pošlo krivo u vašoj aplikaciji. Hvatanjem svih iznimki zapravo u svom kôdu govorite da ste spremni na bilo kakvu grešku i da možete u svojoj aplikaciji napraviti baš sve što je potrebno da se oporavite od svih grešaka. Zar vam se to ne čini besmislenim?

Također, kad se govori o oporavljanju od iznimaka, važno je da se od njih oporavljate na pravi način. Nužno je da, ako ste u mogućnosti, napravite ipak nešto više od pukog ispisivanja poruke o greški. Od vas se očekuje da u situacijama iznimki zatvorite datoteku koja je uzrokovala problem, otkažete daljnje izvršavanje programa koji pokušava kontaktirati nedostupnu bazu podataka ili pomognete korisniku u ispravljanju mogućih grešaka ako je do iznimke došlo njegovim upisom neispravnih podataka.

Veoma je važno i koristiti *finally* blok kad god je to moguće i potrebno. Iako možete *try catch finally* blok napisati bez *finally* bloka i u njemu samo provjeravati kôd i izvršiti jednostavno hvatanje iznimki, veoma je korisna mogućnost koju *finally* nudi. Ponovimo još jednom, korištenjem *finally* bloka osiguravate se da će kôd sadržan u njemu doista i biti izvršen. Ako se dogodi iznimka, to je ključna stvar, jer će se u protivnom prekinuti daljnje izvršavanje, dok ćete korištenjem *finally* bloka još imati na raspolaganju izvršavanje nekoliko naredbi kojima ćete osloboditi korištene resurse i slične stvari. Čak i ako se ne dogodi iznimka, *finally* blok će se izvršiti, pa je tako njegova uloga višestruka.

Hvatanje iznimki nije ovisno o jeziku u kojem je pisan dio programa u kojem se dogodila iznimka, jer su sve iznimke tipa *System.Exception*, a on je ugrađen u sam .NET *framework*. Primjerice, možete u C#-u korištenjem ovdje opisane *try catch finally* sintakse hvatati iznimke koje se dešavaju u metodama pisanim u VB.NET-u: u prethodnom primjeru je metoda *provjeriKorisnika* mogla biti spremljena u posebnom modulu pisanom u VB.NET-u, a to ne bi uopće utjecalo na rad programa – pojavila bi se iznimka i ona bi bila uhvaćena u C# dijelu programa.

Baci bombu, goni iznimku...

Osim što u .NET-u možete hvatati iznimke, vi ih možete i stvarati odnosno *bacati*. To postižete korištenjem *throw* naredbe kojom stvarate novu iznimku i prosljeđujete je ostatku programa. Primjerice, u glavnom dijelu programa imate realiziran *try catch finally* blok u kojem obavljate neke akcije, pozivate različite vlastite funkcije i provjeravate pojavljivanje iznimki.

No kad se u nekoj od pozivanih funkcija dogodi nešto što onemogućava njeno daljnje ispravno izvršavanje (primjerice, proslijeđeni parametri nisu u odgovarajućem obliku), vi možete *baciti* iznimku. Ona će biti uhvaćena u glavnom programu i bit će ispisane detaljnije informacije o greški. Pogledajte sljedeći kôd:

```
void GlavnaMetoda() {
   try
   {
      // uvodni kôd
      provjeriKorisnika("Hrvoje
Horvat");
      // kôd koji očekuje da
određeni korisnik postoji
   }
   catch (NotSupportedException
e)
   {
      // ispis informacije o
greški, oslobađanje resursa...
   }
}
void provjeriKorisnika(string
korisnik) {
   // ako korisnik ne postoji...
   throw new
NotSupportedException("Korisnik
" + korisnik + " ne postoji!");
}
```

U ovom smo primjeru stvorili vlastitu iznimku! Ako u funkciji *provjeriKorisnika* doista bude utvrđeno da korisnik "Hrvoje Horvat" ne postoji i izvrši se *throw* naredba, u *catch* dijelu u glavnoj metodi može se ispisati poruka iznimke korištenjem *e.Message* koja će glasiti "Korisnik Hrvoje Horvat ne postoji!".

Sama sintaksa *throw* naredbe je jednostavna – ona baca novu iznimku, stoga iza nje slijedi *new* i tip iznimke (podsjetimo se, korištenjem *new* operatora stvara se novi objekt određenog tipa). Parametara nove iznimke može biti više, no možete se odlučiti i da samo upišete poruku koju će iznimka sadržavati. Tu poruku zatim možete ispisati iz drugih dijelova programa u kojem tu iznimku uhvatite.

III. dio

Dijelovi .NET-a

8. POGLAVLJE

Windows Forms

U ovom poglavlju:

- Kada pisati prozorske aplikacije
- Kako napraviti aplikaciju za Windowse
- Kontrole u kolekciji klasa Windows Forms
- Manipulacija formama
- Crtanje i ispis na pisač
- Priprema aplikacija za distribuciju

Nekoć, u doba dok je Internet bio tek igračka zaigranih entuzijasta, razvojni alati u Windowsima bavili su se isključivo izradom prozorskih aplikacija. Pod tim pojmom podrazumijevamo sve aplikacije koje kao osnovu sučelja koriste prozore (po kojima su Windowsi i dobili ime) odnosno, kako se oni stručnije nazivaju, forme.

Glavna odlika prozorskih aplikacija je što se one nalaze i pokreću na lokalnom računalu. Takvih aplikacija na svom računalu sigurno imate pregršt – od jednostavnih Notepada i Calculatora, preko pomoćnih programa kao što su Internet Explorer, WinAmp ili Acrobat Reader do velikih aplikacija i paketa poput Microsoft Officea, Corel Drawa i AutoCAD-a.

Prozorske aplikacije najčešće dolaze na instalacijskim CD-ima ili ih, ukoliko se radi o manjim aplikacijama, možete skinuti s Interneta. U oba slučaja potrebno ih je instalirati ili

raspakirati na računalu, a kasnije, kada ih zatrebate, samo ih pokrenete. Najčešće se radi o izvršavanju neke datoteke s nastavkom ".exe", što često puta radimo i ne znajući zahvaljujući prečicama (engl. *shortcut*) koje pokazuju na te datoteke. No sve ste to, vjerujemo, već znali...

Prozorske aplikacije u ovoj knjizi koristimo kao sinonim za aplikacije pisane za Windowse. Takve aplikacije se u stručnoj literaturi nazivaju i *rich client* aplikacije.

Širenjem Interneta i pojavom sve većeg broja brzih i praktički stalnih veza postepeno se smanjuje potreba za prozorskim aplikacijama. Brojne funkcionalnosti nekoć rezervirane isključivo za prozorske aplikacije dostupne su u obliku web-stranica. Primjerice, svom sandučiću elektroničke pošte možete pristupati koristeći prozorsku aplikaciju za *e-mail* ili preko često jednako funkcionalnog web-sučelja. Drugi zgodan primjer su intranetski sustavi u poduzećima, koji sve češće pružaju mogućnosti koje su prije bile domena prozorskih poslovnih aplikacija.

Prozorske aplikacije mogu biti samostojeće (engl. *stand-alone*) ili klijentsko-serverske (engl. *client/server*). Samostojeća aplikacija je ona kojoj su svi podaci i sva logika smješteni na istom računalu. Klijentsko-serverske aplikacije u svom radu zahtijevaju vezu s nekom drugom, serverskom aplikacijom, najčešće smještenom na nekom drugom računalu u mreži.

Naravno, to ne znači da se prozorskim aplikacijama crno piše. Postoji velik broj situacija u kojima one predstavljaju bolje ili čak jedino rješenje. Spomenimo samo računalne igre, multimedijalne svirače, uredske aplikacije, grafičke programe, razne klijente za internetske servise... U krajnjoj liniji, ako ništa drugo, uvijek će postojati preglednik web-stranica koji je i sam prozorska aplikacija.

U ovom ćemo poglavlju naglasak staviti na biblioteku klasa Windows Forms, koja predstavlja svojevrsnu nadogradnju sustava koji se dosad koristio. Naime, za izradu korisničkog sučelja u starijim se razvojnim alatima koristio znatno kompliciraniji Win32 API (kratica za *application program interface*). Osim Windows Formsa, pozabavit ćemo se i klasama u biblioteci GDI+ koja omogućava crtanje po ekranu. Krenimo redom...

Forme i kontrole

Vjerujemo da ste se u sučelju Windowsa toliko udomaćili da ni ne primjećujete njegove sastavnice. Prva stvar koju treba uočiti je forma. Forme su osnova korisničkog sučelja i možemo ih poistovjetiti sa prozorima.

Korisničko sučelje gradimo tako da na formu postavljamo kontrole. Kontrole su zajedničko ime za natpise, gumbe, kućice za upis, padajuće liste i razne druge oblike prezentacije informacija i mogućnosti. Velik broj kontrola predefiniran je na nivou operativnog sustava, no moguće je dodavati nove kontrole, modificirati postojeće te stvarati vlastite.

Peripetije u pozadini

Forma nije ništa drugo nego objekt stvoren instanciranjem određene klase. Sjetimo se – klasa je nacrt prema kojem se stvara (instancira) objekt. Klasa na temelju koje stvaramo standardnu formu je System.Windows.Forms.Form, što možemo vidjeti iz sljedećeg primjera:

```
public class Form1 :
System.Windows.Forms.Form
```

Sad postaje kompliciranije. Svaka forma je ujedno i klasa. Drugim riječima, stvaranjem objekta forme stvaramo i novu klasu. Nova klasa nasljeđuje sve karakteristike osnovne klase (dakle, klase System.Windows.Forms.Form) i ako na njoj ne napravimo nikakvu promjenu, one će biti identičnih karakteristika. Međutim, promijenimo li našoj formi karakteristike (mijenjamo svojstva, dodajemo kontrole), mijenjat će se i karakteristike naše klase i ona će se razlikovati od osnovne koju smo inicijalno naslijedili.

Zamislimo sada da u programu imamo potrebu za novom formom. Imamo dvije mogućnosti – kreirati novu formu na temelju klase System.Windows.Forms.Form ili na temelju klase naše izmijenjene forme. Napravimo li to na prvi način, dobit ćemo praznu formu (jer klasa System.Windows.Forms.Form sadrži nacrt za praznu formu). Ako pak naslijedimo našu klasu, nova će forma imati sve karakteristike koje ona sadrži – sva svojstva, sve kontrole i sve ostalo:

```
public class Form2 :
NasaAplikacija.Form1
```

A što je s klasama? Kada kreiramo instancu kontrole na formi, koristimo klasu koja je opisuje i kreiramo objekt, no ne i klasu:

```
this.label1 = new
System.Windows.Forms.Label();
this.button1 = new
System.Windows.Forms.Button();
```

Za svaku kontrolu postoji druga klasa, no svima je zajedničko da nasljeđuju osnovnu klasu Control; ona sadrži najosnovnije karakteristike koje kontrola mora imati. Štoviše, i klasa System.Windows.Forms.Form inicijalno nasljeđuje klasu Control, pa možemo reći da je i forma – kontrola.

Kao što smo već vidjeli u četvrtom poglavlju, na izgled i ponašanje formi i kontrola utječemo mijenjanjem njihovih svojstava i povezivanjem funkcionalnosti na događaje koje stvaraju. Kako kontrola

ima puno, a svojstava, metoda i događaja još više, čak i površno šetanje kroz njih zauzelo bi prilično prostora. Zato smo se odlučili upoznati vas s kontrolama kroz nekoliko jednostavnih i složenijih primjera. Ne samo da ćete upoznati velik dio kontrola i njihova svojstva već ćemo zajedno proći kroz proces izrade aplikacija.

Primjer: Beskorisna aplikacija

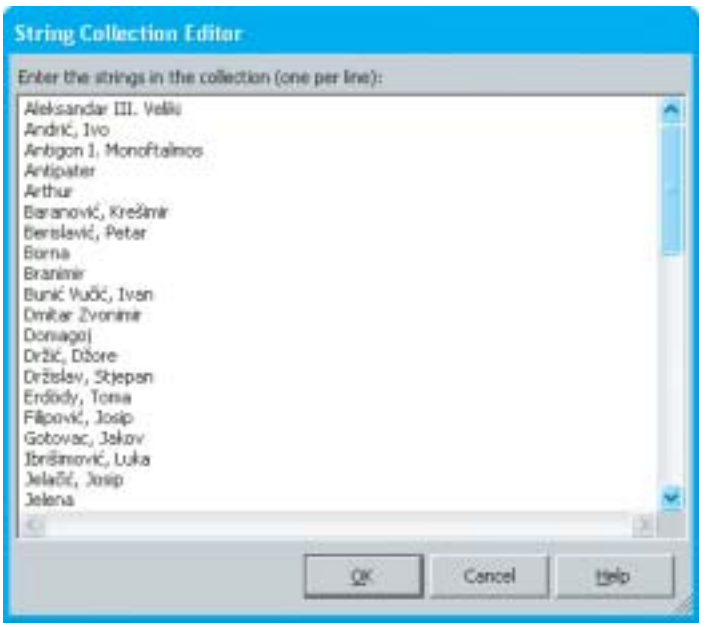
Za početak ćemo napraviti beskorisnu aplikaciju za unos podataka. Kažemo "beskorisno" jer se podaci u njoj neće pamtiti pa će, jednom kada iz nje izađete, svi uneseni podaci biti izgubljeni. Ipak, naučit ćemo kako složiti jednostavno korisničko sučelje, koristiti gumbe, polja za upis, padajuće liste i još mnogo toga.

Slika 8-1: ***Beskorisna aplikacija u dizajnerskom načinu***

Čitanje knjiga u krevetu ili na plaži je super, no preporučujemo vam da dok prolazite kroz ovo i sljedeća poglavlja budete za računalom, imate otvoren Visual Studio .NET 2003 i isprobavate sve što spominjemo. Tako se puno bolje uči.

Kreirajmo novi projekt za prozorsku aplikaciju i na praznu formu dovucimo sljedeće kontrole: TextBox, ComboBox, ListBox i dvije kontrole tipa Button. Cilj nam je napraviti sljedeće: dozvoliti korisniku da upiše neki izraz u polje za upis, odabere neku vrijednost iz padajuće liste te klikom na gumb doda kombinaciju tih dvaju izraza na listu. Preostali gumb poslužit će nam za demonstraciju zatvaranja prozora odnosno izlaska iz aplikacije.

Prvo moramo promijeniti nekoliko svojstava. Kontroli textBox1 svojstvo Text postavite na praznu vrijednost (izbrišite postojeće), dok gumbima isto svojstvo postavite na vrijednosti "dodaj" za button1 i "zatvori" za button2. Oko padajuće liste ćemo imati nešto više posla – prvo ju je potrebno napuniti podacima. To ćemo napraviti tako da u redu svojstva Items kliknemo na gumbić s tri točkice, koji će se pokazati nakon što taj red označimo. Klikom na njega otvorit će se prozor s velikim poljem za upis. U njega upisujemo ponuđene vrijednosti – svaku vrijednost u svoj red, kao što možemo vidjeti na slici 8-2. Konačno, svojstvo DropDownStyle postavite na DropDownList.



Slika 8-2: ***Izbore u padajućem izborniku upisujemo svaki u poseban red.***

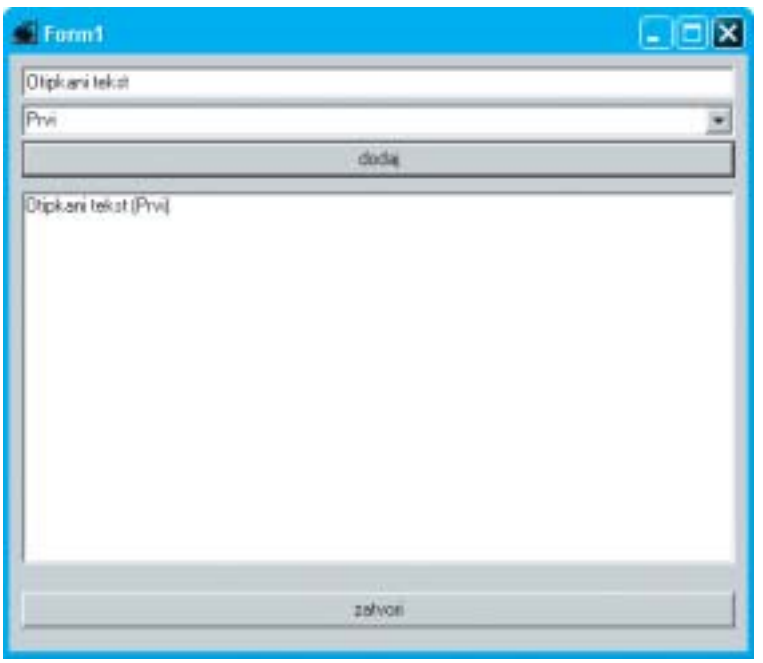
Sljedeći korak je dodavanje funkcionalnosti gumbima. Prvi gumb (button1) služit će za prebacivanje unesenog i odabranog teksta u listu. Stoga trebamo dvokliknuti na njega i automatski će se kreirati funkcija unutar koje ćemo napisati kôd koji će napraviti ovo što smo upravo spomenuli. Istu smo stvar mogli napraviti i na drugi način – označiti gumb, u pomoćnom prozoru kliknuti na simbol munje

kako bismo ušli u događaje vezane uz taj gumb te dvokliknuti na bijeli dio reda u kojem piše Click. Međutim, kako je događaj Click glavni događaj za kontrolu tipa gumb, dovoljno je na njega dvokliknuti. (Isto vrijedi i za ostale kontrole, samo što one mogu imati neki drugi glavni događaj.)

Kôd koji ćemo dodati u tako dobivenu funkciju glasi ovako:

```
listBox1.Items.Add(textBox1.Text + " (" + comboBox1.Text + ")");
```

Što taj komad kôda radi? Kontrola tipa ListBox ima zadaću držati i prikazivati listu izraza. Ta lista izraza smještena je u kolekciju Items pa kada joj želimo pristupiti pišemo izraz listBox1.Items. Kolekcija Items sadrži metodu Add pomoću koje joj dodajemo novi zapis u kolekciju (odnosno novi izraz na listu). Sve što navedemo kao parametar te metode (u zagradi) bit će dodano na kraj liste.



Slika 8-3: ***Beskorisna aplikacija prilikom izvršavanja***

Izrazom textBox1.Text pristupamo vrijednosti koja je upisana u polje za unos textBox1, dok izrazom comboBox1.Text pristupamo vrijednosti koja je odabrana u padajućoj listi. Spajanjem tih dvaju izraza i dodavanjem zagrada iz kozmetičkih razmaka kreiramo izraz koji ćemo dodati u listu. Drugim riječima, ukoliko je u polju za upis upisan izraz "Otipkani tekst", a iz padajuće liste odabrana vrijednost "Prvi" onda će redak u listi nakon pritiska na gumb izgledati ovako: "Otipkani tekst (Prvi)". Uostalom, pokrenite aplikaciju (F5) i isprobajte!

Preostaje nam još dodati funkcionalnost drugom gumbu, nazvanom button2. Kôd koji ćemo kod njega dodati nakon dvoklika je sljedeći:

```
Close();
```

Taj komadić kôda nalaže formi u kojoj se nalazi da se zatvori, a pošto je naša forma osnovna (i jedina) forma naše aplikacije, zatvaranjem te forme zatvorit će se i cijela aplikacija. Uočite da prije metode Close ne navodimo kontrolu na koju se ona odnosi – stoga što se odnosi na formu.

Ukoliko aplikaciju želimo zatvoriti iz forme koja nije osnovna, koristit ćemo sljedeći kôd:

```
Application.Exit();
```

Ovaj primjer poslužit će nam da malo bolje upoznamo kontrole koje u njemu koristimo, njihova svojstva, događaje i metode.

Vizualna svojstva forme

Želite li da vam aplikacije budu dosadne, sive, jednolične i nimalo atraktivne? Trebali biste, jer prozorska aplikacija nije mjesto za pokazivanje umjetničkih sklonosti, no ipak ćemo vam pokazati kako izmijeniti neke vizualne karakteristike forme. Napominjemo još jednom – nemojte pretjerivati – aplikacije moraju biti jednostavne, oku ugodne i pregledne.

Dio vizualnih svojstava koje postavimo formi naslijedit će i sve kontrole koje ona sadrži. Primjerice, promijenimo li formi svojstvo za pozadinsku boju, nju će naslijediti i gumbi.

Boje i pozadina

Formi možemo mijenjati boju pozadine. To radimo mijenjajući svojstvo BackColor kojem valja pridružiti vrijednost tipa System.Drawing.Color. Njemu možemo pridružiti boje iz tri svojevrsna "spremišta" koje Visual Studio nudi u padajućem izborniku boja, no mi ćemo se upoznati s njima na nešto dubljem nivou pošto se koriste na mnogo mjesta, ne samo u ovoj kontroli nego i ostalima. Prvo "spremište" je skupina sistemskih boja u Windowsima koje su zapisane u klasi System.Drawing.SystemColors. Drugim riječima, kada napišemo sljedeću vrijednost:

```
System.Drawing.SystemColors.Control
```

dobit ćemo boju koja je zapisana u postavkama Windowsa kao *defaultna* boja za površinu prozora u *appletu* "Display Properties". Na taj način poštujemo postavke korisnika koje je podesio

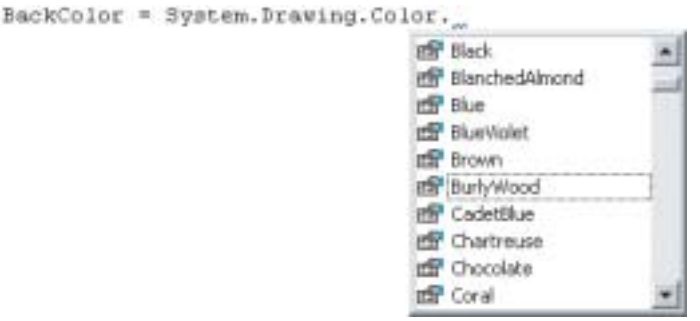
na nivou operativnog sistema. Sve kontrole inicijalno imaju podešene boje na ovaj način i bez zaista jakog razloga ne biste to trebali dirati.

Svako svojstvo kontrole određenog je tipa i kod pridruživanja vrijednosti moramo joj pridružiti vrijednost upravo tog tipa. Neka svojstva su numeričkih vrijednosti (najčešće *int*), druga logičkih (*boolean*), a dobar dio svojstava je nekog posebnog tipa definiranog baš za to svojstvo. Na ovo valja paziti samo kod pridruživanja vrijednosti u kodu, a ako to radite pomoću Visual Studijevog sučelja, on će se sam za to pobrinuti.

Međutim, postoji mogućnost da želite podesiti neku konkretnu boju, nevezanu uz postavke operativnog sustava. Velikom broju boja možete pristupiti preko pamtljivih imena ispred kojih treba navesti ime klase u kojoj su definirane. Evo nekoliko primjera:

```
System.Drawing.Color.Black;
System.Drawing.Color.White;
System.Drawing.Color.Blue;
System.Drawing.Color.BurlyWood;
```

Slika 8-4:
IntelliSense u akciji – izbor boja



Dakako, imena boja ne morate pamtiti. Dovoljno je u kodu napisati ime klase (System.Drawing.Color) i otipkati znak točke – IntelliSense će se pobrinuti za ostalo.

Treći način definiranja boja predviđen je za slučaj kada ne postoji predefinirano ime za boju koju želite koristiti. U tom slučaju, valja znati njenu RGB-vrijednost i napisati je u sljedećem obliku:

```
System.Drawing.Color.FromArgb(255, 128, 0);
```

RGB-vrijednosti boja sastoje se od tri broja. Prvi označava udio crvene boje (*red*), drugi zelene (*green*), a treći plave (*blue*). Brojevi moraju biti tipa *byte*, što znači da su u rasponu od 0 do 255. Vrijednost (0, 0, 0) predstavlja crnu, a (255, 255, 255) bijelu boju. Kratica RGB dolazi od prvih slova engleskih naziva boja.

Osim pozadine, boja se na formi koristi i za svojstvo ForeColor. Ono služi za definiranje boje teksta i grafike na formi. Kako forma sama po sebi ne sadrži nikakav tekst, do izražaja dolazi nasljeđivanje svojstava koje smo maloprije spomenuli – boja koju definirate svojstvu ForeColor neće biti aplicirana na formu, nego na sve kontrole koje se na njoj nalaze, osim onih koje za to svojstvo imaju definiranu drugu boju.

Kao primjer evo kôda koji valja ubaciti u funkciju vezanu uz događaj klika na gumb (button1_Click), i koji će učiniti da forma promijeni boju nakon klika na gumb:

```
BackColor = System.Drawing.Color.BlueViolet;
```

Osim boje, u pozadinu možemo smjestiti i sliku. To se radi postavljajući svojstvo BackgroundImage, za koje vrijedi isto upozorenje kao i za mijenjanje boja – samo ako morate.

Pozicija i veličina forme

Svojstvo Location određuje gdje će forma na ekranu biti pozicionirana. Ako, kao u našem primjeru, aplikacija ima samo jednu formu, onda ovo svojstvo određuje poziciju u odnosu na cijeli ekran. S druge strane, ukoliko se forma nalazi unutar neke druge forme (slučaj kakav ćemo upoznati nešto kasnije u poglavlju), onda se svojstvo Location odnosi relativno na tu roditeljsku formu. Ista je stvar i s kontrolama – koordinate u svojstvu Location odnose se na površinu forme na kojoj se kontrola nalazi.

Lokaciju ćemo najlakše definirati pomoću sučelja Visual Studija, no ako osjetite potrebu za promjenom ili čitanjem informacije o lokaciji forme u kodu, tom ćete svojstvu pristupiti ovako:

```
int kooX = Location.X;  // čitanje koordinate X
int kooY = Location.Y;  // čitanje koordinate Y

// premještaj forme na lokaciju 100, 200
Location = new System.Drawing.Point(100, 200);
```

Primjećujete kako je repozicioniranje forme (ili kontrole općenito) prilično složeno. To je stoga što je svojstvo Location tipa System.Drawing.Point. Taj tip podataka je vrijednosni tip, što znači da vraća samu vrijednost, a ne i referencu na što se ona odnosi. Zato nije moguće napisati:

```
Location.X = 100;  // pogrešno!
```

Stoga postoje dva dodatna svojstva koja nam omogućavaju jednostavnije pozicioniranje formi. Sljedeći primjer ima isti efekt kao i zadnja linija prošlog (ispravnog) primjera:

```
Left = 100;
Top = 200;
```

Slična je priča i sa svojstvima koja određuju veličinu forme. Pristupanje i mijenjanje kroz svojstvo Size slično je primjeru sa svojstvom Location:

```
int velX = Size.Width;   // čitanje širine
int velY = Size.Height;  // čitanje visine

// određivanje veličine forme na 500 x 600 piksela
Size = new System.Drawing.Size(500, 600);
```

Jednostavnije pisano, zadnji bi red izgledao ovako:

```
Width = 500;
Height = 600;
```

Osim spomenutih svojstava, postoje još svojstva Right i Bottom. Svojstvo Right jednako je zbroju svojstava Left i Width, dok je svojstvo Bottom jednako zbroju svojstava Top i Height. Ti odnosi među svojstvima su stalni, pa ako, primjerice, smanjite vrijednost Height za sto, i vrijednost Bottom će se smanjiti za isto toliko.

Pokazivač miša

Pokazivača miša korisnici računala prihvaćaju zdravo za gotovo. Sasvim je logično da on u nekim situacijama bude u obliku strelice, ponekad u obliku ruke, a ponekad u obliku za pisanje teksta. Naravno, vi to u svom programu možete kontrolirati, pa i promijeniti izgled pokazivača miša na vašoj formi.

To možete učiniti mijenjajući svojstvo Cursor odnosno pridružujući mu vrijednosti iz klase System.Windows.Forms.Cursors. Sučelje Visual Studija, kao i kod boja, nudi padajući izbornik za promjenu za vrijeme dizajniranja aplikacije, a ako to želite napraviti za vrijeme izvršavanja aplikacije, koristit ćete izraz poput ovoga:

```
Cursor = System.Windows.Forms.Cursors.WaitCursor;
```

Kako na početku datoteke uključujemo *namespace* System.Windows.Forms, gornji redak možemo pisati i kraće:

```
Cursor = Cursors.WaitCursor
```

Zaglavlje forme

Svaki standardni prozor ima zaglavlje u kojem se nalaze ikona i naslov prozora s lijeve te kućice za minimiziranje, maksimiziranje, normalno stanje prozora i njegovo gašenje s desne strane. Pogledajte bilo koju klasičnu prozorsku aplikaciju i znat ćete o čemu pričamo.

Sva je ta svojstva moguće mijenjati odnosno utjecati na njihovo postojanje. Ikonu pridružujemo mijenjanjem svojstva Icon, na sličan način kao i pozadinsku sliku, dok je naslov prozora smješten u svojstvu Text.

Slika 8-5: ***Zaglavlje forme (slijeva nadesno) – ikona forme (s odgovarajućim padajućim izbornikom), naslov, ikone za minimiziranje, maksimiziranje i zatvaranje***

Dostupnost kućica za minimiziranje i maksimiziranje podešavamo pomoću svojstava MinimizeBox i MaximizeBox (pridružujemo im vrijednosti tipa *boolean*), a na isti način uključujemo odnosno isključujemo kućicu s upitnikom (za dozivanje sustava pomoći; svojstvo HelpButton). Zanimljivo je da kod svojstava vezanih uz minimiziranje i maksimiziranje forme ne isključujemo samo kućice

u zaglavlju, nego općenito dozvoljavamo ili nedozvoljavamo minimiziranje odnosno maksimiziranje forme. Naime, osim preko tih kućica, te je radnje moguće izvršiti preko izbornika koji se otvara klikom na ikonu prozora ili desnim klikom na ime prozora u *taskbaru*.

Želite li sve muhe ubiti jednim udarcem, iskoristite svojstvo ControlBox. Postavljanjem vrijednosti tog svojstva na *false* ugasit ćete sve "sličice" u zaglavlju (ikonu i sve kućice s desne strane), a ostaviti samo naslov prozora.

Ponašanje i izgled forme

Nekoliko svojstava utječe na izgled forme. Tu treba spomenuti svojstvo FormBorderStyle koje definira koji tip obruba će forma imati. Osim vizualnog prikaza forme, kroz to svojstvo definiramo i hoće li korisnik programa moći mijenjati veličinu forme (*resize*) razvlačenjem ruba prozora. U tablici 8-1 možete vidjeti vrijednosti (pobrojane tipove) koje to svojstvo može poprimiti.

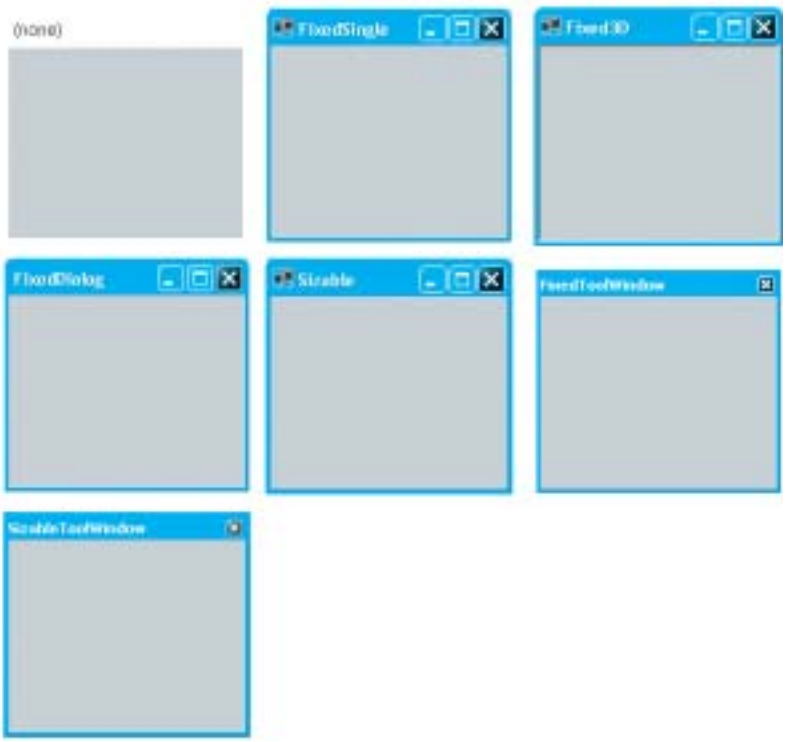
Tablica 8-1:
Moguće vrijednosti za svojstvo FormBorderStyle

Vrijednost	Opis
FormBorderStyle.Fixed3D	fiksni trodimenzionalni rub
FormBorderStyle.FixedDialog	debeli fiksni rub u stilu dijaloškog okvira
FormBorderStyle.FixedSingle	tanki fiksni rub, debeo samo jedan piksel
FormBorderStyle.FixedToolWindow	fiksni rub tipa pomoćnog prozora (engl. *tool window*), koji se ne pojavljuje u *taskbaru* niti kada korisnik pritisne Alt+Tab
FormBorderStyle.None	bez ruba
FormBorderStyle.Sizable	trodimenzionalni rub; prozoru je moguće mijenjati dimenzije
FormBorderStyle.SizableToolWindow	promjenjivi pomoćni prozor, koji se ne pojavljuje u *taskbaru* niti kada korisnik pritisne Alt+Tab.

Kao što piše u tablici, pomoćni prozori se ne pojavljuju u *taskbaru*. Međutim, ako želimo da se i prozor nekog drugog tipa tamo ne pokazuje, možemo to napraviti mijenjajući vrijednost svojstvu ShowInTaskbar, za vrijeme dizajniranja forme ili za vrijeme izvršavanja aplikacije.

Pomoću svojstva StartPosition određujemo gdje će se forma prilikom aktivacije pojaviti. Ukoliko pridružimo vrijednost FormStartPosition.Manual, forma će se smjestiti na parametre određene svojstvom Location. FormStartPosition.CenterScreen nalaže prikaz forme na centru ekrana, FormStartPo-

sition.CenterParent u centru roditeljske kontrole (o tome nešto kasnije), a FormStartPosition.WindowsDefaultLocation i FormStartPosition.WindowsDefaultBounds na mjesto definirano u Windowsima, s razlikom da potonji ignorira veličinu prozora definiranu svojstvom Size.



Slika 8-6: ***Usporedba različitih stilova formi (svojstvo FormBorderStyle)***

Formi možemo odrediti da se otvara u maksimiziranom odnosno minimiziranom stanju, pa svojstvo iz prethodnog odlomka neće imati smisla. To će se dogoditi ako svojstvu WindowState dodijelimo vrijednost FormWindowState.Minimized odnosno FormWindowState.Maximized. To svojstvo možemo mijenjati i za vrijeme izvršavanja aplikacije, pa tako možemo našem sveprisutnom događaju klika na gumb dodati liniju koja slijedi, i na taj način maksimizirati formu:

```
WindowState = FormWindowState.Maximized;
```

Konačno, prozoru možemo uključiti svojstvo TopMost koje će ga držati na vrhu svih prozora, bez obzira na to što nije trenutno aktivan prozor. Ta je mogućnost korisnicima poznata pod imenom “Always on top”.

Kako bismo isprobali tu mogućnost, dodat ćemo sljedeću liniju kôda (znate već gdje):

```
TopMost = ! (TopMost);
```

Ta će linija omogućiti da pritiskom na gumb uključujemo i isključujemo svojstvo forme da bude iznad svih prozora ne samo svoje, već i ostalih aplikacija. Ukoliko je to svojstvo uključeno (*true*), znak uskličnika će ga pretvoriti u neistinitu logičku vrijednost (*false)* i zatim tu vrijednost pridružiti svojstvu. I obrnuto.

Prozirne i šuplje forme

Iako ih rijetko susrećemo u praksi (što nije nužno loše), prozirne i šuplje forme vrlo je jednostavno napraviti. Prozirnost forme definiramo parametrom Opacity, kojem navodimo postotnu vrijednost prozirnosti forme. *Defaultna* vrijednost je 100%, što znači da forma neće biti prozirna.

Šupljinu u formi određujemo pomoću boje. Naime, svojstvu TransparencyKey pridružujemo neku boju, a ono se brine da svako pojavljivanje te boje u formi rezultira – rupom. Kroz tu rupu u prozoru možete vidjeti i kliknuti ono što bi inače bilo prekriveno prozorom. Postavimo li vrijednost tog svojstva na bijelu boju, naš će primjer dobiti rupu na tekstualnom polju za upis, padajućem izborniku i listi.

Slika 8-7: ***Forma nekonvencionalnog oblika napravljena modifikacijom našeg primjera***

Prozirne forme rijeko imaju smislen razlog postojanja, no šupljima se već može naći korisnih implementacija. Naime, one se često koriste za kreiranje prozora nekonvencionalnih oblika, poput prozora sa zaobljenim rubovima ili forme u obliku kruga.

Na slici 8-7 možete vidjeti jednu takvu formu, napravljenu na temelju primjera Beskorisne aplikacije. Recept je sljedeći: prvo valja napraviti sliku koju ćemo staviti u pozadinu forme. Jedna od boja na toj formi (po mogućnosti neka kričava, koja se sigurno neće pojavljivati na formi) bit će zamijenjena "rupom", tako da ćete nju najvjerojatnije staviti na rub slike. Ostatak slike, ono što će biti vidljivo, obojite nekom drugom bojom, najbolje sivom, koja se i inače koristi za pozadinu prozora.

Tako pripremljenu sliku pridružite svojstvu BackgroundImage. Zatim svojstvu TransparencyKey pridružite onu kričavu boju sa slike kako bi taj dio prozora postao šupalj. Konačno, svojstvu FormBorderStyle pridružite vrijednost None kako biste uklonili rub forme i zaglavlje, koji bi uništili cijeli posao.

Olakšajmo potrebnima!

Kako bi i osobe s invaliditetom mogle koristiti računala, postoje posebni dodaci za operativne sustave koji im u tome pomažu. Neki od tih dodataka nalaze se i u Windowsima, a neke je potrebno naknadno instalirati.

Ta se dodatna programska podrška ne oslanja samo na operativni sustav, već i na aplikacije. Tako i vaša aplikacija može biti prilagođena za korištenje od strane osoba s invaliditetom, i to podešavajući tri jednostavna svojstva koja se nalaze na svim vizualnim kontrolama.

Svojstvo AccessibleName treba sadržavati kratko ime koje korisnika može asocirati na funkciju kontrole. Nešto duži opis kontrole valja upisati u svojstvo AccessibleDescription, dok kao vrijednost svojstva AccessibleRole treba izabrati ulogu te kontrole. U većini slučajeva potonje svojstvo nije potrebno mijenjati s početne vrijednosti Default jer će kroz tu vrijednost sistem za svaku kontrolu vraćati njenu pravu ulogu.

Odnos kontrola i forme

Kao što smo već više puta ponovili, forma može sadržavati razne kontrole. One se na formi ne nalaze samo vizualno, nego to pravilo slijedi i objektni model. Sve se kontrole, naime, nalaze u kolekciji Controls. Primjerice, želimo li nekoj od njih promijeniti svojstvo Text, to možemo učiniti i ovako:

```
Controls[0].Text = "novo svojstvo Text";
```

Kolekcija Controls odnosi se na formu, no ime forme nije potrebno navoditi jer se sav kôd nalazi u klasi (formi) na koju se odnosi. Indeks "0" označava prvu kontrolu na formi, no često je teško

znati koja je to. Naime, prva kontrola je ona koja se prva kreira, što u praksi ne znamo iako možemo vidjeti u skrivenom dijelu kôda (regija Windows Form Designer generated code). To je ona kontrola nad kojom je prva pozvana metoda Controls.Add. U praksi to izgleda ovako:

```
this.Controls.Add(this.button2);  // kontrola s indeksom 0
this.Controls.Add(this.button1);  // kontrola s indeksom 1
this.Controls.Add(this.listBox1); // kontrola s indeksom 2
```

Ovakvo korištenje kontrola je prekomplicirano, pa postoji direktno referenciranje kontrola koje smo već susreli:

```
button2.Text = "novo svojstvo Text"
```

Zanimljivo je da osim što preko forme možete doći do kontrola, i preko kontrola možete doći do forme kojoj ona pripada. Evo kako preko kontrole promijeniti svojstvo koje pripada formi:

```
button2.TopLevelControl.Text = "Novi naslov forme";
```

Dakako, istu je stvar moguće dobiti osjetno kraćim izrazom, no samo smo željeli pokazati kako su forma i njezine kontrole povezane. Osim gornjega, postoje još perverzniji primjeri, kao što je sljedeći koji preko kontrole pristupa formi da bi promijenio svojstvo toj istoj kontroli:

```
button2.TopLevelControl.Controls[0].Text = "Novi natpis na button2";
```

Nakon što smo objasnili objektnu povezanost forme i njezinih kontrola, red je da se pozabavimo nešto manje apstraktnim stvarima.

Pozicija, veličina i skrivanje kontrola

Pozicija kontrola relativna je u odnosu na unutarnji dio forme (znači, ne računaju se rubovi i zaglavlje forme). Drugim riječima, ako neka kontrola stoji na poziciji 100, 200, to znači da je ona 100 piksela desno i 200 piksela dolje od unutarnjeg gornjeg lijevog ruba. U većini slučajeva nećete u kodu mijenjati pozicije i veličine kontrola već ćete ih vizualno postaviti u dizajnerskom načinu. Ipak, ukoliko vam zatreba, sintaksa je jednaka kao kod pozicioniranja forme (jer, sjetimo se, i forma je kontrola):

```
int kooX = button1.Location.X;  // čitanje koordinate X
int kooY = button1.Location.Y;  // čitanje koordinate Y

// premještaj kontrole button1 na lokaciju 100, 200
button1.Location = new System.Drawing.Point(100, 200);
```

```
// drugi način premještaja kontrole
button1.Left = 100;
button1.Top = 200;
```

Ista je stvar i s veličinom kontrola:

```
int velX = button1.Size.Width;   // čitanje širine
int velY = button1.Size.Height;  // čitanje visine

// određivanje veličine kontrole na 200 x 40 piksela
button1.Size = new System.Drawing.Size(200, 40);

// drugi način određivanja veličine kontrole
button1.Width = 200;
button1.Height = 40;
```

Kontrole na formi možete i skrivati. To činite postavljanjem svojstva Visible na *true* (kontrola će biti vidljiva) odnosno na *false* (kontrola će biti nevidljiva). Imajte na umu da ovo utječe isključivo na vizualnu karakteristiku kontrole – ona i dalje postoji i sva njezina svojstva se mogu normalno koristiti.

Ukoliko vam se ne da natezati s koordinatama i veličinom kontrole, možete joj reći da bude zalijepljena cijelom dužinom uz neki od rubova. To radimo svojstvom Dock, koje možemo postaviti na vrijednosti None, Top, Bottom, Left, Right i Fill.

Razvlačenje formi

Korisnici su navikli da veličinu prozora (forme) mogu prilagoditi svojim željama i potrebama. Međutim, ako netko razvuče formu iz našeg primjera, dobit će samo velik, neiskorišten prostor.

Ne zaboravite, da bi forma mogla biti razvlačena, mora biti adekvatno podešeno svojstvo FormBorderStyle!

Da se to ne bi događalo, kontrole imaju svojstvo Anchor. Ono je inicijalno postavljeno na vrijednosti (Top | Left). Zbog takvih postavki događa se da sve kontrole prilikom razvlačenja ostanu u gornjem (Top) lijevom (Left) uglu forme. Želimo li da sve budu vezane uz donji desni ugao, svima ćemo svojstvo Anchor postaviti na vrijednosti (Bottom | Right). Ako navedemo dvije suprotne vrijednosti, prim-

jerice (Left | Right), onda kontrola neće mijenjati lokaciju, nego veličinu – bit će razvučena u tom smjeru.

Slika 8-8:
Beskorisno razvučena Beskorisna aplikacija

Prvo ćemo se pozabaviti vodoravnim razvlačenjem. Ako korisnik razvuče formu u širinu, želimo da sve kontrole također budu razvučene u širinu tako da razmak između kontrole i desnog ruba bude uvijek isti. Zato ćemo svim kontrolama svojstvo Anchor podesiti na vrijednosti (Top | Left | Right). Time smo odredili da se prilikom razvlačenja forme u širinu i sve kontrole na njoj proporcionalno prošire.

Želite li neko svojstvo promijeniti na više kontrola odjednom, označite sve takve kontrole (držite tipku Ctrl prilikom označavanja) i onda izmijenite željeno svojstvo. Promjena će biti aplicirana na sve kontrole koje su bile označene. Pripazite da sve kontrole imaju svojstvo koje želite mijenjati!

Preostaje nam rješavanje okomitog razvlačenja. Ukoliko tu primijenimo isti princip, kontrole će se početi razvlačiti jedna preko druge i sučelje će biti uništeno. Zato trebamo odrediti jednu centralnu kontrolu koja će zauzeti višak površine, dok će ostale samo mijenjati poziciju, a ne i visinu. Takav je princip apliciran kod svih aplikacija u Windowsima – razvlačite li prozor Microsoft Worda, ulogu centralne kontrole imat će središnji, bijeli dio u koji unosimo tekst.

Slika 8-9: ***Beskorisna aplikacija s pravilno podešenim svojstvom Anchor***

Pravilo koje vrijedi u većini slučajeva jest – najveća kontrola dobiva ulogu centralne. U našem primjeru to je kontrola listBox1. Dalje je jednostavno – centralnoj kontroli dodjeljujemo sve četiri vrijednosti svojstva Anchor – (Top | Left | Right | Bottom). Kontrole koje se nalaze iznad nje dobivaju vrijednosti (Top | Left | Right), dok onima ispod nje pridružujemo (Left | Right | Bottom). Rezultat možete vidjeti na slici 8-9. (Preporučujemo vam da ovo svojstvo podešavate na kraju dizajniranja korisničkog sučelja, jer svoje ponašanje pokazuje i u dizajnerskom načinu, što može biti smetnja.)

Razvlačenje formi u aplikacijama je zgodna stvar, no može postojati granica do koje želite korisniku dozvoliti smanjivanje (ili povećavanje) forme. Primjerice, forma veličine 10x10 piksela nema nikakvog smisla, jer kontrole neće biti niti vidljive, a kamoli upotrebljive. Stoga vam biblioteka Windows Forms omogućava postavljanje gornje i donje granice dimenzija forme. One se kriju iza svojstava MaximumSize i MinimumSize i istog su tipa kao i svojstvo Size. Ukoliko su ta svojstva postavljena na (0, 0), granice neće postojati.

Osim na razvlačenje, svojstvo MaximumSize utječe i na maksimiziranje forme. Drugim riječima, ukoliko maksimizirate formu koja ima postavljeno svojstvo MaximumSize, ona će biti povećana do granice definirane u tom svojstvu, a ne preko cijelog ekrana.

Gumbi potvrde i poništenja

Ako ste se malo više igrali s našim primjerom, onda ste sigurno poželjeli da nakon upisa i odabira parametara ne morate u ruke uzimati miša, nego da umjesto klika na gumb možete istu stvar napraviti pritiskom tipke Enter na tipkovnici. Ništa lakše! Treba u svojstvima forme potražiti svojstvo AcceptButton i iz padajuće liste izabrati gumb koji želimo da bude pritisnut kada korisnik klikne tipku Enter. U našem slučaju to će biti button1.

Istu stvar možete napraviti i sa svojstvom CancelButton, jedino što on neće reagirati na tipku Enter, već na tipku Esc. U našem primjeru tom svojstvu pridružen je gumb button2, tako da se pritiskom tipke Esc zatvori forma i završava program.

Tipkovnička šetnja sučeljem

Osim tipaka Enter i Esc, vrlo često korištena tipka za navigaciju poljima za unos je i tipka Tab odnosno kombinacija Shift + Tab. Njihovu namjenu već znate – prelazak u sljedeće odnosno prethodno polje na stranici – a kako ih implementirati u vlastiti program, saznat ćete sada.

Zapravo, neprecizno je govoriti o implementaciji, jer je samim kreiranjem sučelja cijela stvar već funkcionalna. Vi možete i morate podesiti stvar da radi u skladu s očekivanjima korisnika. Uzmimo za primjer našu Beskorisnu aplikaciju – nije logično da nas nakon tekstualnog polja za upis tipka Tab prebaci na gumb ili listu. Trebamo postići da se prebacivanje vrši na vizualno prvu sljedeću kontrolu kako bi ponašanje sučelja bilo intuitivno i očekivano. To radimo mijenjajući vrijednost svojstva TabIndex svakoj kontroli. Kontroli za koju želimo da prva ima fokus, odmah nakon pokretanja programa, dodijelit ćemo vrijednost 0. Sljedećoj ćemo pridružiti vrijednost 1, onoj nakon nje 2 i tako dalje.

Ukoliko ne dodijelite vrijednosti svojstvima TabIndex, tipka Tab koristit će onaj redoslijed kojim su kontrole dodavane u kolekciju Controls. To u većini slučajeva nije dobro rješenje, pa ovoj funkcionalnosti svakako posvetite pažnju.

Naravno, ne može se do svih kontrola doći tipkom Tab. Neke od njih za tim jednostavno nemaju potrebe jer ne pružaju nikakvu interaktivnu funkcionalnost (najbolji primjer za takve kontrole su one tipa Label, o kojima ćemo nešto kasnije), dok nekima i sami možete isključiti tu funkcionalnost. Tome služi svojstvo TabStop kojem treba pridružiti vrijednost tipa *boolean*. Sve kontrole to svojstvo imaju inicijalno postavljeno na *true,* što znači da im je pristup tipkom Tab omogućen.

Gumbi

Nakon sage o formama, vrijeme je da se pobliže pozabavimo kontrolama koje smo koristili u primjeru Beskorisne aplikacije. Jedna od najvažnijih kontrola je ona tipa Button. Vjerujemo da vam funkciju gumba u aplikaciji ne moramo objašnjavati, no zanimljivo je proći neka njegova svojstva, kako biste ga u svojim aplikacijama mogli bolje iskoristiti.

Svojstva koja spominjemo kod neke kontrole mogu vrijediti i za druge kontrole. Priručno pravilo glasi: ako se svojstva isto zovu, onda imaju istu ili vrlo sličnu funkciju. Kod nekih ćemo kontrola spomenuti svojstva koja "dijeli" s već spomenutim kontrolama, no kod većine nećemo, jer bi zbog velikog broja svojstava takva nabrajanja bila nepregledna i nepraktična. Popis svih svojstava dostupnih kod neke kontrole možete pronaći u dokumentaciji ili u sučelju Visual Studija.

Odluku o stilu gumba morate donijeti na početku izrade aplikacije. Naime, vrlo je važno da vam svi gumbi budu istog stila, kako bi aplikacija dobila na preglednosti. Stil gumba mijenjate mijenjajući svojstvo FlatStyle, kojem možete dodijeliti četiri vrijednosti (iz pobrojanog tipa System.Windows.Forms.FlatStyle): Standard, Flat, Popup i System. Usporedbu prvih triju tipova možete vidjeti na slici 8-10, dok četvrti predstavlja jedan od njih, ovisno o postavkama sistema.

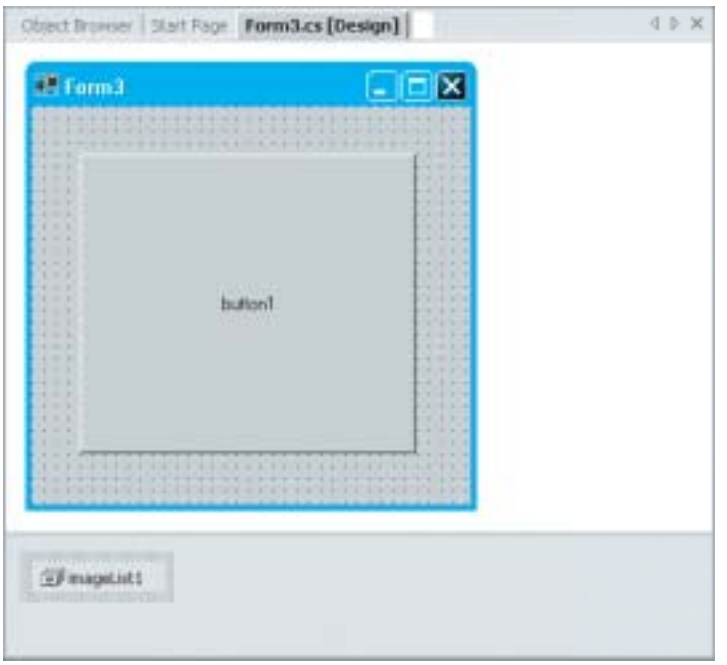
Svaki gumb ima natpis kojim korisniku daje do znanja čemu služi. Taj se natpis mijenja svojstvom Text. Poziciju natpisa u gumbu možete podesiti mijenjajući svojstvo TextAlign. Na taj način natpis možete smjestiti u sredinu gumba (inicijalna postavka), uz neki od rubova ili u neki od kutova. Tip i veličinu slova kojima je ispisan natpis možemo mijenjati preko svojstva Font iako to nije uobičajeno.

Slika 8-10: ***Usporedba triju stilova gumba u tri različita stanja – izaberite koji želite, samo budite dosljedni!***

Osim natpisa, gumb može imati i sličicu. Nju postavljamo tako da svojstvu Image dodijelimo sliku, na isti način kao što smo radili s formom. Poziciju slike na gumbu, slično kao što smo radili s tekstom, mijenjamo svojstvom ImageAlign.

Lista slika

Međutim, često je potrebno na istom gumbu imati više slika. Primjerice, jednu kada je gumb u stanju mirovanja, drugu kada se nad njime nalazi pokazivač miša i treću kada je pritisnut. Kako to napraviti?

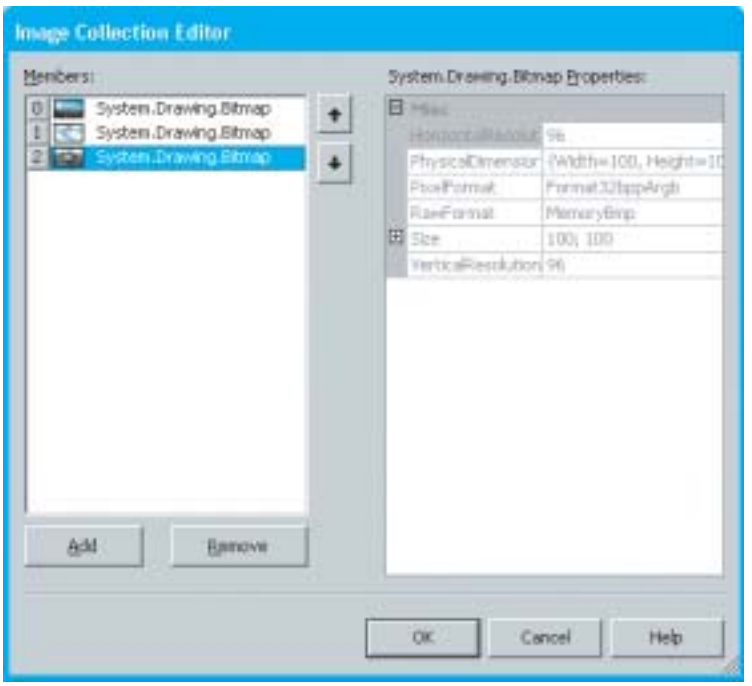


Slika 8-11: ***Nevizualne kontrole prikazuju se izvan forme.***

Da bismo to postigli, moramo na formu dovući kontrolu tipa ImageList. Ta kontrola ne spada među vizualne kontrole, pa se neće pojaviti na samoj formi, nego u području ispod nje (vidi sliku 8-11).

Kontrola ImageList predstavlja svojevrsno spremište slika. Svaka slika koju dodamo u tu kontrolu bit će sastavni dio aplikacije i moći ćemo je koristiti u mnogim slučajevima, uključujući i naš primjer s gumbom koji se upravo spremamo napisati.

U listu sliku treba smjestiti onoliko slika koliko vam treba, a u našem slučaju će to biti tri slike. Bitno je da pripazite da slike budu istih dimenzija kao i gumb – ne zato što drugačije ne bi radilo, nego zato što će tako ljepše izgledati.



Slika 8-12: ***Pomoćni prozor za dodavanje slikovnih datoteka u listu slika***

Slike dodajete preko svojstva Images. Pritiskom na gumbić s tri točkice otvorit će vam se prozor (slika 8-12) u kojem ćete slike dodavati klikom na gumb Add. Taj prozor krije i jednu vrlo važnu stvar – indekse slika u kolekciji. Kao i kod svih ostalih kolekcija, prva slika ima indeks 0. Zapamtite koja slika ima koji indeks, kako biste ih kasnije znali referencirati. Najbolje je smisliti i slijediti neki (vama) logičan princip. Primjerice, mi ćemo staviti tri slike – prvu (indeks 0) za gumb u stanju mirovanja, drugu za gumb ispod pokazivača miša i treću za pritisnut gumb.

Nakon što smo zatvorili pomoćni prozor, trebamo podesiti još neka svojstva liste slika. Nikako ne smijemo zaboraviti svojstvo ImageSize pomoću kojeg definiramo veličinu slika. Inicijalna vrijednost je

16x16 piksela, što u praksi znači da će vaše slike, bez obzira na originalnu veličinu, biti reducirane na tu dimenziju. To, dakako, nije poželjno, pa stoga tu vrijednost podesite na dimenziju slika koje ste dodali.

Osim dimenzija možete podesiti dubinu (broj) boja svojstvom ColorDepth i prozirnost slike odabirom neke boje u svojstvu TransparentColor.

Vrijeme je da napravljenu listu slika povežemo gumbom. To činimo tako da među svojstvima gumba nađemo ono nazvano ImageList. Tamo će se u padajućoj listi pojaviti naša lista slika (*defaultnog* imena imageList1) koju trebamo izabrati. Naravno, sučelje ne zna koju sliku iz liste želimo postaviti kao osnovnu, pa stoga u svojstvo ImageIndex trebamo upisati vrijednost indeksa slike. Radi se o svojstvu tipa *int*, no sučelje nam pomaže u obliku malih inačica slika, koje se kriju iza pojedinog indeksa kako bismo lakše izabrali. Mi odabiremo indeks 0, kako bismo prvu sliku postavili za osnovnu.

Događaji vezani uz miša

Za ostatak funkcionalnosti morat ćemo posegnuti za malo kôda. To ćemo učiniti tako da za određene događaje na gumbu vežemo funkcije koje će na njemu mijenjati sliku. Za početak, prilikom prelaska pokazivačem miša želimo da se prikaže slika s indeksom 1. U pomoćnom prozoru Properties prelazimo na popis događaja (ikona žute munje), pronalazimo događaj MouseEnter, dvoklikamo i otvara nam se funkcija koja treba izgledati ovako:

```
private void button1_MouseEnter(object sender, System.EventArgs e)
{
    button1.ImageIndex = 1;
}
```

Događaj MouseEnter nastupa kada pokazivač miša uđe u vidljivo područje kontrole na koju se odnosi, u našem slučaju kontrole button1. Promjenom indeksa u svojstvu ImageIndex, automatski se mijenja i slika prikazana na gumbu.

Prilikom nastupa nekog događaja (primjerice MouseEnter), svaka kontrola ima i ugrađenu metodu (OnMouseEnter) koja će se automatski izvršiti. Uz nju, mi možemo napisati vlastitu funkciju vezanu uz taj događaj (koja će se automatski nazvati ImeKontrole_Enter). Uvijek se izvršavaju obje funkcije, i naša i ugrađena.

Kada se pokazivač miša makne s gumba, treba vratiti prvu sliku. Ovaj se puta vežemo na događaj MouseLeave:

```
private void button1_MouseLeave(object sender, System.EventArgs e)
{
    button1.ImageIndex = 0;
}
```

U slučaju pritiska tipke miša nastupa događaj MouseDown, a kada tipku otpustimo, nastupit će događaj MouseUp. Evo kôda koji valja zakačiti na te događaje:

```
private void button1_MouseDown(...)
{
    button1.ImageIndex = 2;
}

private void button1_MouseUp(...)
{
  button1.ImageIndex = 1;
}
```

Kad se pritisne tipka miša, prikazat će se treća slika (indeks 2) iz liste slika. Kada se tipka otpusti, prikazuje se druga slika (indeks 1). Zašto druga, pitate se, kad je prva osnovna slika? Zato što se u trenutku otpusta tipke pokazivač miša još uvijek nalazi nad gumbom, pa je potrebno vratiti onu sliku koja se tada prikazuje, a to je u našem slučaju druga slika.

Događaji koje smo upravo obradili postoje u svim vizualnim kontrolama; osim njih, postoje još dva događaja vezana uz miša i njegove dogodovštine na ekranu. Prvi među njima je MouseMove koji se događa svaki put kada se miš pomakne nad kontrolom. Kod korištenja tog događaja vrlo je važno imati na umu da može nastupiti izuzetno velik broj puta, posebno ako korisnik miša pomiče vrlo sporo ili kruži unutar jedne te iste kontrole. Smještanje čak i jednostavnog kôda u funkciju vezanu uz taj događaj može rezultirati značajnim trošenjem sistemskih resursa.

Da biste dobili dojam koliko često događaj MouseMove nastupa, povežite na njega sljedeću funkciju:

```
private void button1_MouseMove(...)
{
          Text += ".";
}
```

Ta će funkcija svaki put kada bude pozvana dodati jednu točku u naslov prozora. Primijetite kako će se brzo točke nakupiti!

Zadnji događaj iz paketa “mišjih” događaja je MouseHover, koji nastupa u slučaju da se korisnik neko kraće vrijeme zadrži nad kontrolom ništa ne radeći. Najbolji primjer tog događaja (iako se

ne rješava pomoću njega) jest onaj mali, žuti pravokutnik koji se u brojnim aplikacijama pojavljuje nad ikonama u alatnoj traci da nam u nekoliko riječi opiše funkciju koja se krije iza ikone.

Polje za unos teksta (jedna linija)

Sljedeća kontrola na listi za proučavanje je TextBox. Kao što nezgrapni prijevod u naslovu kaže, radi se o kontroli koja korisniku omogućava upis niza znakova.

Kontrola TextBox koristi se i za jednolinijske i višelinijske upise teksta. Na ovom mjestu proučavamo osnovna svojstva vezana uz jednolinijsku funkcionalnost, dok ćemo se svojstvima vezanim uz tekstualno polje za upis s više linija baviti u sljedećem primjeru.

Osnovne stvari u radu s ovom kontrolom vidjeli smo u primjeru Beskorisna aplikacija. Znamo da se uneseni tekst sprema u svojstvo Text i kako mu se pristupa.

Sad ćemo se malo pozabaviti vizualnim karakteristikama polja za unos teksta. Za početak, moguće je postaviti tri stila okvira mijenjajući svojstvo BorderStyle. Dostupni su nam *defaultni* trodimenzionalni okvir (vrijednost Fixed3D), jednostavan okvir s rubom debelim jedan piksel (FixedSingle) i polje za unos bez okvira (None).

Tip i veličinu slova (svojstvo Font), kao i kod ostalih kontrola, možete mijenjati i kontrolama tipa TextBox, a da ne biste trebali ručno podešavati veličinu kontrole, postoji svojstvo AutoSize. Kada je njemu pridružena vrijednost *true*, visina kontrole automatski će se prilagoditi odabranom tipu slova.

Broj znakova koje korisnik unosi možete limitirati mijenjajući svojstvo MaxLength. Želite li da korisnik može unositi isključivo mala ili isključivo velika slova, svojstvo CharacterCasting postavit ćete na jednu od vrijednosti iz pobrojanog tipa System.Windows.Forms.CharacterCasing (Lower ili Upper umjesto Normal).

Kada želite da korisnik ne vidi slova koja upisuje (primjerice, kod unosa lozinke), u svojstvo PasswordChar unijet ćete znak (samo jedan!) koji će se pokazivati na ekranu bez obzira na to koje je slovo pritisnuto. Uvriježilo se da taj znak bude zvjezdica (*).

Ponekad postoji potreba za onemogućavanjem unošenja ili mijenjanja teksta u polju. To, dakako, možete riješiti skrivanjem kontrole mijenjajući svojstvo Visible, no puno elegantnije i preglednije rješenje se krije iza svojstva ReadOnly. Ako je ono postavljeno na *true*, korisniku neće biti dozvoljena nikakva promjena u polju za upis, a cijelo će polje biti zasivljeno.

Događaji vezani uz promjene svojstava

U priči o poljima za unos teksta javlja se prilika da se malo više pozabavimo događajima koji nastupaju kad se neko svojstvo promijeni. Stvar je vrlo jednostavna – primjerice, u trenutku kada se promijeni vrijednost svojstva Text, nastupit će događaj TextChanged. Mi za njega, kao i za svaki drugi događaj, možemo vezati funkciju i tako dodati neku funkcionalnost.

Događaji koji nastupaju nakon promjene nekog svojstva ne postoje za sva svojstva. Koja su svojstva dostupna za koju kontrolu, potražite u dokumentaciji ili sučelju Visual Studija.

U sljedećem primjeru napravit ćemo da svakom promjenom teksta u polju za upis promijenimo i ime forme:

```
private void textBox1_TextChanged(...)
{
    Text = textBox1.Text;
}
```

Padajuća lista

Ukoliko korisniku ne želite omogućiti slobodan upis neke informacije već želite da izabere iz liste ponuđenih opcija, koristit ćete kontrolu ComboBox. Njezino glavno svojstvo je DropDownStyle, kojim određujete izgled i ponašanje kontrole. To svojstvo tipa System.Windows.Forms.ComboBoxStyle inicijalno ima vrijednost DropDown koja osim izbora među ponuđenim opcijama nudi i mogućnost upisivanja vlastite vrijednosti. Ako ne želite korisniku dozvoliti slobodan unos, svojstvo DropDownStyle postavite na vrijednost DropDownList. Treća moguća vrijednost (Simple) rijetko se koristi. Po funkcionalnosti je slična prvoj, osim što ne postoji strelica za otvaranje padajuće liste, već kontrola na formi izgleda kao polje za unos teksta, a kroz ponuđene opcije korisnik se mora kretati strelicama na tipkovnici.

Već smo pokazali kako pomoću Visual Studijevog sučelja dodati opcije u padajuću listu (svojstvo Items). Sada nam preostaje pokazati kako opcije dodavati i mijenjati u kodu. Želimo li dodati dodatnu opciju, koristit ćemo sljedeću sintaksu:

```
comboBox1.Items.Add("nova opcija");
```

Primjećujete – stvar izgleda isto kao i dodavanje nove stavke u listu koju smo koristili u primjeru Beskorisna aplikacija. Pažljiviji čitatelji petog poglavlja prepoznat će u svojstvu Items kolekciju *stringova,* s kojom možemo raditi sve što smo u tom poglavlju naučili – dodavati, brisati, mijenjati i prebrojavati. Evo nekoliko primjera:

```
// brisanje svih članova iz kolekcije
comboBox1.Items.Clear();

// dodavanje novog člana umetanjem na treću poziciju
comboBox1.Items.Insert(2, "Novi član kolekcije");

// brisanje člana s indeksom 2
comboBox1.Items.RemoveAt(2);

// brisanje člana s određenom vrijednosti
comboBox1.Items.Remove("Novi član kolekcije");
```

Padajuća lista može sadržavati neograničen broj članova, a koliko će ih odjednom biti vidljivo na ekranu ovisi o vrijednosti svojstva MaxDropDownItems. To, naravno, ne znači da ostali članovi neće biti dostupni za izbor, nego da ćemo, da bismo do njih došli, morati skrolati listu.

Kontrola padajuća lista omogućava i automatsko sortiranje svojih članova. Ukoliko želimo da budu sortirani, svojstvo Sorted postavit ćemo na *true*. Imajte na umu da ovim svojstvom možete unijeti zbrku u indekse unutar kolekcije, jer, primjerice, posljednji dodani član neće nužno biti na posljednjem mjestu, već na onom kojem mu pripada u sortiranom nizu. Međutim, ukoliko promijenite nekome od članova ime, kolekcija neće automatski biti sortirana, već će član ostati na mjestu na kojem se nalazio. Želite li takvu kolekciju ipak sortirati, možete napraviti sljedeće:

```
comboBox1.Sorted = false;
comboBox1.Sorted = true;
```

Spomenuti trik ne biste smjeli prečesto koristiti jer je sortiranje kolekcije na taj način prilično sporo, a o sortiranju većih kolekcija na ovaj način nemojte niti razmišljati.

Kada korisnik odabere jednu od ponuđenih vrijednosti, željet ćete joj pristupiti. Bez obzira je li ju ručno upisao ili izabrao među ponuđenima, ona će biti smještena u svojstvo Text. Međutim, postoji i svojstvo SelectedItem koje će sadržavati izraz iz kućice samo ako je on izabran među ponuđenima, a ako ga je korisnik ručno upisao, to će svojstvo biti prazno (prazan *string*).

Želite li znati indeks odabrane vrijednosti, koristit ćete svojstvo SelectedIndex. Osim čitanja, u to svojstvo možete i pisati te na taj način u kodu promijeniti izbor. Primjerice, želite li podesiti izbor na prvi član kolekcije, napisat ćete sljedeći izraz:

```
comboBox1.SelectedIndex = 0;
```

Često puta nećete znati indeks člana kolekcije koji želite postaviti kao odabrani. U tom će vam slučaju pomoći metoda FindString. Evo kako odabrati prvog člana koji počinje znakovima “abc”:

```
comboBox1.SelectedIndex = comboBox1.FindString("abc")
```

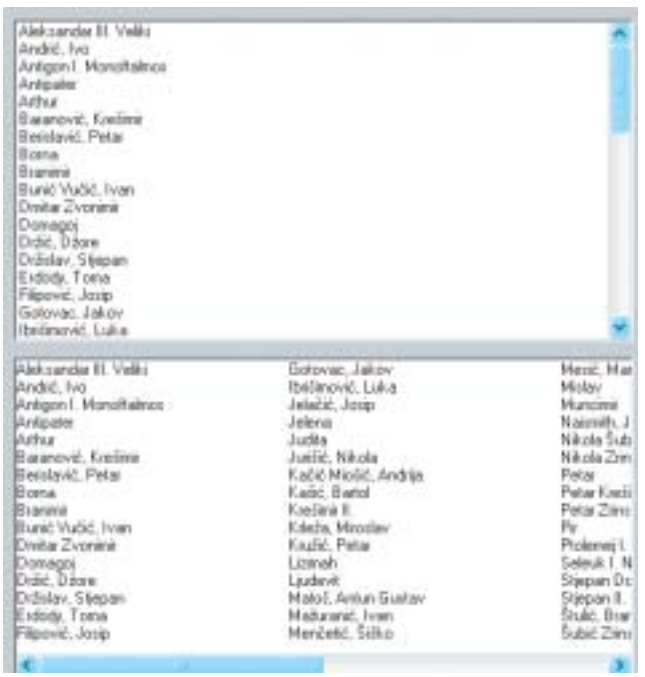
Kao što iz primjera možete zaključiti, metoda FindString vraća indeks prvog člana u kolekciji koji počinje navedenim nizom znakova.

Lista

Kontrole tipa ListBox u mnogočemu su slične padajućim listama zbog istog načina pohrane svojih članova. Sve što smo rekli vezano uz kolekciju Items, vrijedi i ovdje, samo što se članovi kolekcije na ekranu drugačije prezentiraju.

Svojstvo BorderStyle određuje tip ruba kojim će lista biti uokvirena. Dostupne su vrijednosti None (bez ruba), FixedSingle (rub širine jednog piksela) i Fixed3D (trodimenzionalan rub).

Ukoliko ima više članova kolekcije nego što je veličina kontrole, pojavit će se tzv. *scrollbarovi*. Ako želite da oni uvijek budu prisutni na ekranu, svojstvo ScrollAlwaysVisible postavite na *true*.



Slika 8-13: ***Usporedba kontrole ListBox s isključenim (gore) i uključenim (dolje) svojstvom MultiColumn***

Članovi kolekcije Items na ekranu se mogu prikazivati na dva načina – sa stupcima i bez njih. Kako to izgleda na ekranu možete vidjeti na slici 8-13. Svojstvo koje određuje način prikaza naziva se MultiColumn i može poprimiti vrijednost *true* odnosno *false*. Ukoliko je prikaz po stupcima uključen, ima smisla podesiti i vrijednost svojstva ColumnWidth. Njime određujemo širinu stupca, a ukoliko to ne učinimo (odnosno, vrijednost svojstva bude 0), širina svakog stupca bit će automatski podešena ovisno o dužini članova kolekcije.

Namjena kontrole ListBox može biti puko prikazivanje liste, no korisniku možemo omogućiti i odabir jednog ili više članova. Namjenu kontrole određujemo svojstvom SelectionMode, a vrijednosti koje mu možemo pridružiti su None, One, MultiSimple i MultiExtended. Prva vrijednost ne dozvoljava označavanje članova, druga pruža mogućnost označavanja samo jednog, dok ostale omogućavaju odabir više članova istovremeno.

U slučaju da odaberete vrijednost One, označenoj vrijednosti ćete moći pristupiti preko svojstava SelectionIndex i SelectionItem. Prvo svojstvo sadržavat će indeks odabranog člana u kolekciji, dok će SelectionItem sadržavati njegovu vrijednost.

Treba li vam u programu funkcionalnost odabira više članova liste odjednom, morate odabrati između vrijednosti MultiSimple i MultiExtended. Razlika među njima svodi se na način na koji korisnik odabire članove. U prvom načinu klik na neoznačeni član učinit će ga označenim, dok će klik na označeni rezultirati neoznačenim. Ipak, u korisničkim sučeljima češće se koristi drugi način, koji omogućava označavanje više članova tek uz korištenje tipki Shift i Ctrl ili potezanjem miša pritisnute tipke, baš kao što je to napravljeno u samim Windowsima.

Većem broju označenih članova ne možemo pristupiti na isti način. Za te slučajeve postoje svojstva SelectionIndices i SelectionItems. To su kolekcije koje sadrže sve označene članove, prva njihove indekse u originalnoj kolekciji, a druga njihove vrijednosti.

I jednom označenom članu možete pristupati preko svojstava SelectionIndices i SelectionItems. U tom će slučaju te kolekcije imati samo jednog člana s indeksom 0.

Uzmimo za primjer da nam treba kôd koji će sve označene članove iz kontrole listBox1 prebaciti u comboBox1. To možemo napraviti na dva načina. Prvo, korištenjem svojstva SelectedItems:

```
comboBox1.Items.Clear();  // praznimo comboBox1
for (int n = 0; n < listBox1.SelectedItems.Count; n++)
{
   object vrijednostOznacenog = listBox1.SelectedItems[n];
```

```
    comboBox1.Items.Add(vrijednostOznacenog);
}
```

Varijablu n vrtimo u petlji *for* sve dok je ona manja od ukupnog broja članova u kolekciji listBox1.SelectedItems. Unutar petlje varijabli vrijednostOznacenog tipa *object* pridružujemo jednu od vrijednosti iz kolekcije, onu koja ima indeks n. U sljedećem redu tu vrijednost dodajemo u comboBox1.

Istu smo stvar mogli napisati puno kraće, korištenjem naredbe *foreach*:

```
comboBox1.Items.Clear();
foreach (object vrijednostOznacenog in listBox1.SelectedItems)
{
    comboBox1.Items.Add(vrijednostOznacenog);
}
```

Drugi, nešto složeniji način je onaj u kojem koristimo svojstvo SelectedIndices:

```
comboBox1.Items.Clear();
for (int n = 0; n < listBox1.SelectedIndices.Count; n++)
{
    int indeksOznacenog = listBox1.SelectedIndices[n];
    comboBox1.Items.Add(listBox1.Items[indeksOznacenog]);
}
```

U ovom načinu iz kolekcije listBox1.SelectedIndices dobivamo indekse označenih članova. Stoga unutar petlje koristimo varijablu indeksOznacenog tipa *int*. U redu dodavanja kao parametar metodi Add nećemo navesti samo varijablu, nego ćemo pomoću nje iz originalne kolekcije listBox1.Items izvući označene članove. I ovaj se način može osjetno skratiti korištenjem naredbe *foreach*, no to vam ostavljamo za samostalan rad.

Svojstva SelectionIndices i SelectionItems moguće je samo čitati. Za definiranje označenih članova koristite metodu SetSelected.

Želite li iz kôda označiti odnosno odznačiti članove kolekcije listBox1.Items, dobro će vam doći metoda SetSelected. Počnimo jednostavnim primjerom – označimo prva tri člana kolekcije:

```
listBox1.SetSelected(0, true);
listBox1.SetSelected(1, true);
listBox1.SetSelected(2, true);
```

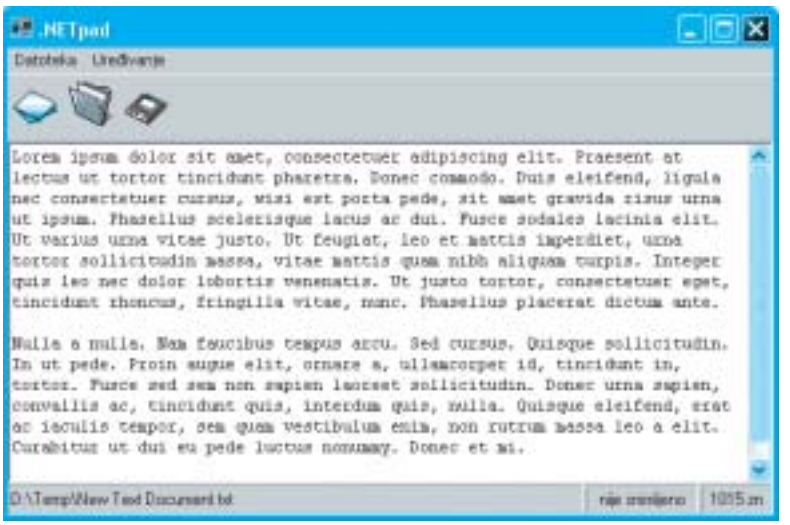
Iz primjera vidimo da metoda SetSelected zahtijeva dva parametra – indeks člana i vrijednost tipa *boolean* koja kazuje hoće li taj član biti označen (*true*) ili neoznačen (*false*).

Isprobajmo to na mrvicu složenijem primjeru, u kojem je cilj odznačiti sve članove kolekcije:

```
for (int n = 0; n < listBox1.Items.Count; n++)
{
   listBox1.SetSelected(n, false);
}
```

Primjer: Tekstualni editor

Nakon što smo iskoristili Beskorisnu aplikaciju kako smo znali i mogli, red je da napravimo novi primjer, pomoću kojega ćemo upoznati neke nove kontrole, svojstva i događaje.



Slika 8-14: ***Ovo nam je cilj – primjer bi na kraju trebao izgledati ovako***

Kao što možete vidjeti na slici 8-14, ne radi se o sasvim jednostavnoj aplikaciji. Sučeljem dominira veliko tekstualno polje za upis (TextBox, ovaj puta višelinijski), a oko njega su se smjestili

izbornik (MainMenu), traka s alatima (ToolBar) i statusna traka (StatusBar). Otvorite novi projekt i dovucite spomenute kontrole. Traka s alatima će se automatski smjestiti na vrh, statusna traka na dno, a izbornik će zasada biti isključivo izvan forme.

Polje za unos teksta (višelinijsko)

Prvo ćemo krenuti s podešavanjem polja za unos. Kako ćemo ga koristiti za uređivanje tekstualnih datoteka, moramo ga pretvoriti u višelinijsko polje za upis. To radimo mijenjanjem vrijednosti svojstva MultiLine na *true*. Nakon toga tu kontrolu možete razvući tako da zauzima sav prostor između dviju traka. Osim toga, svojstvo ScrollBars valja postaviti na Both kako bismo uključili *scrollbarove* koji će nam kod dužih tekstova omogućiti da vidimo i onaj dio koji ne stane na ekran.

Svojstvo WordWrap sigurno već prepoznajete po imenu. Radi se o opciji koja odlučuje hoće li se tekst u polju lomiti na kraju retka (kao što, primjerice, čini Word) ili izlaziti nadesno, izvan vidljivog dijela kontrole (kao što je uobičajeno kod editiranja programskog kôda).

Kod višelinijskih polja važno je uključiti (postaviti na *true*) svojstva AcceptsReturn i AcceptsTab. Naime, ako su ona isključena, korisnik neće moći koristiti tipke Enter i Tab unutar teksta, već će njihovo stiskanje rezultirati pritiskom glavnog gumba na formi (Enter) odnosno prelaskom na sljedeću kontrolu (Tab). Kako se te dvije tipke vrlo često koriste prilikom uređivanja teksta (za prelazak u novi red i uvlačenje), takvo ponašanje ne bi bilo praktično.

U primjeru smo izmijenili i font kojim će slova u polju za unos biti ispisana. Naime, kod tekstualnih smo editora navikli na korištenje *monospaced* fonta pa je red da i naš primjer koristi takav (Courier New; 9pt).

Kod korištenja višelinijskog polja za unos teksta cijeli njegov sadržaj, baš kao i kod jednolinijskog, bit će zapisan u svojstvu Text. Međutim, želite li pristupati svakoj pojedinoj liniji, koristit ćete svojstvo Lines. To svojstvo nije ništa drugo nego kolekcija *stringova* tako da sve rečeno o njoj vrijedi i ovdje.

Ovaj tip polja potencijalno može imati potrebu primiti veliku količinu podataka tako da nam inicijalna vrijednost MaxLength od 32767 znakova vjerojatno neće biti dovoljna. Stoga trebamo to svojstvo postaviti na najveću moguću vrijednost odnosno broj 2147483647 (naime, to svojstvo je tipa Int32). To će nam omogućavati editiranje tekstualnih datoteka veličine do 2 GB!

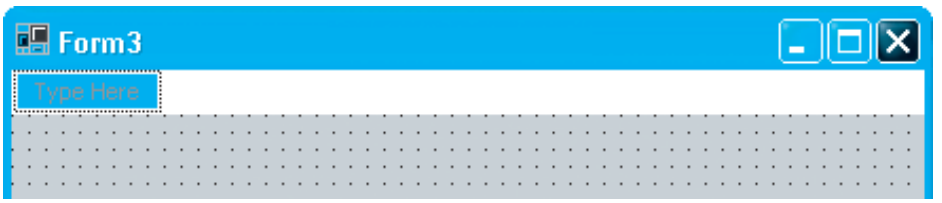
Izbornik

Radite li iole napredniju aplikaciju, radi lakšeg ćete snalaženja u sučelju koristiti izbornik. Dodavanje izbornika u aplikaciju izuzetno je jednostavno jer je sva njegova funkcionalnost sadržana u kontroli MainMenu.

Nakon dodavanja kontrole na radnu površinu, valja krenuti s dodavanjem izbora. Nakon što je označite, na vrhu forme pojavit će se prazan izbornik s natpisom “Type Here” (slika 8-15) i ne ostaje vam drugo

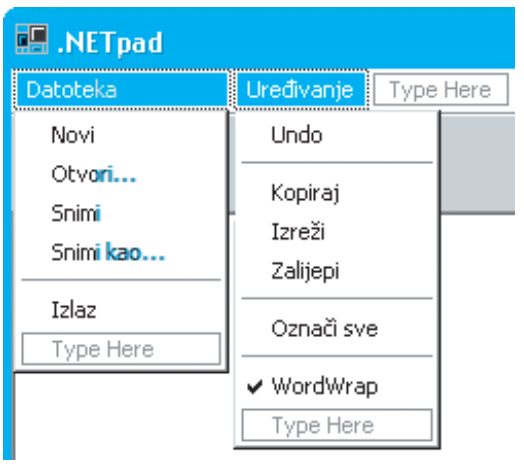
nego učiniti što kaže – počnete tipkati. Tako ćete napraviti svoju prvu stavku u izborniku, a nakon pritiska na tipku Enter, otvorit će se još mjesta s natpisom "Type Here" gdje možete dodavati nove stavke. Stavku po stavku i sagradit ćete izbornik poput ovoga na slici 8-16. Uočite da se tako unesena vrijednost zapisuje u svojstvo Text.

Slika 8-15:
Netom dodana kontrola MainMenu



Uobičajeno je da se stavkama dodjeljuju tipkovničke kratice, kojima se može pristupiti kombinacijom Alt + slovo. Ako želite neko od slova pretvoriti u kraticu tog izbornika, prije njega, u svojstvu Text, stavite znak "&". Primjerice, ako pridružimo vrijednost "Dat&oteka", to će u aplikaciji biti napisano "Datoteka", a tipkovnička kratica će biti Alt + O.

Slika 8-16:
Izbornik primjera Tekstualni editor u dizajnerskom načinu (ne pokušavajte ovo kod kuće, istovremeno otvaranje obje stavke je montirano)



Želite li u izborniku dodati liniju koja će odjeljivati dvije skupine opcija, kao tekst stavke upišite crticu ("-").

Svaka stavka izbornika predstavlja zaseban objekt klase MenuItem, koji se automatski nazivaju po formuli menuItem1, menuItem2, menuItem3... Kako tako nazvani objekti u kasnijem pisanju kôda mogu biti izuzetno nepraktični, odmah vam preporučujemo da ih preimenujete upisujući novu vrijednost u "svojstvo" Name. Mi smo ih u primjeru nazivali logično – stavka s tekstom Novi se zove menuNovi, Otvori je menuOtvori, Snimi je menuSnimi itd. Naravno, ime ne može sadržavati razmake, točke i neke druge specijalne znakove, što ovaj put ne uključuje tzv. hrvatske znakove; njihovo korištenje nije uobičajeno, pa smo, primjerice, stavku Uređivanje nazvali menuUredjivanje.

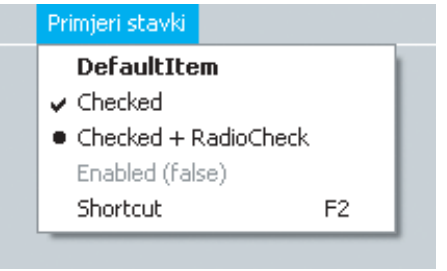
Primijetili ste da svaka stavka može imati podstavke. U tom odnosu, stavka koja ima podstavke naziva se roditelj (engl. parent), a podstavka se zove dijete (engl. child).

Svojstva stavaka

Sam izbornik nema svojstava koje biste željeli mijenjati, no stavke imaju. Ovdje treba naglasiti da svaka stavka predstavlja samostalan objekt pa ako želite svima promijeniti isto svojstvo, morate to napraviti za svaku posebno.

Želite li promijeniti neko svojstvo na više stavaka odjednom, prilikom označavanja istih držite tipku Ctrl. Ovo vrijedi samo za stavke unutar iste roditeljske stavke.

Kao i brojne druge objekte, svaku stavku možete sakriti mijenjajući svojstvo Visible. Ipak, u praksi se za tu svrhu češće koristi svojstvo Enabled, koje ne uklanja stavku iz izbornika, nego samo onemogućava njezino korištenje (stavka postaje "zasivljena").



Slika 8-17:
Stavke raznih tipova na jednom mjestu

Ukoliko stavka služi za uključivanje i isključivanje neke opcije ili mogućnosti, uključit ćete joj svojstvo Checked. Uključenost tog svojstva pokazuje kvačica na stavci, a kada je svojstvo isključeno, stavka izgleda kao i svaka druga. Ukoliko vam se kvačica ne sviđa, možete staviti kružić, uključivanjem svojstva RadioCheck. Ova dva svojstva su isključivo vizualne prirode – funkcionalnost uključivanja i isključivanja kvačice morat ćete napisati sami (kao što ćemo mi napraviti za nekoliko stranica na našem primjeru).

Jednu od stavaka možete proglasiti *defaultnom* uključujući svojstvo DefaultItem. Takvoj će stavci ime biti ispisano podebljanim slovima, a bit će i pokrenuta, ako dvokliknete na njezinu roditeljsku stavku. Primjerice, da na izborniku poput ovoga na slici 8-17 dvokliknemo na roditeljsku stavku "Primjeri stavki" a da je prije ne otvorimo, rezultat bi bio isti kao da smo kliknuli na stavku "DefaultItem". Ova se mogućnost vrlo rijetko koristi.

Svojstvo DefaultItem možete uključiti i za više stavaka pod istom glavnom stavkom. Tada će sve stavke biti podebljane, no samo prva će reagirati na dvoklik.

Jedno od najkorisnijih svojstava stavaka je mogućnost dodjeljivanja direktnih tipkovničkih kratica (tu ne mislimo na one u kombinaciji s tipkom Alt). To činimo mijenjajući svojstvo Shortcut (tip System.Windows.Forms.Shortcut). Ako je svojstvo ShowShortcut uključeno, onda će ta kratica biti prikazana uz stavku, no mnogo je važnija činjenica da se kontrola brine i za dostupnost ovako definirane tipkovničke kratice i kada je izbornik zatvoren. Drugim riječima, dodjeljivanjem tipkovničke kratice nekoj stavci automatski ste riješili postojanje kratice na nivou cijele forme (ali ne i aplikacije!).

Dodjeljivanje funkcionalnosti stavkama

I sami pogađate – klik na neku od stavaka stvara događaj Click, baš kao i klik na bilo koju drugu komponentu. Kao kod gumba, i ovdje je vrlo važno da te događaje povežete s funkcijama i u njima napišete neki kôd jer korisnici očekuju da svaka stavka nečemu služi.

Vratimo se na naš primjer i izbornik koji smo u njemu kreirali. Prvo ćemo programirati najjednostavnije stavke, a prva među njima je svakako stavka Izlaz. Dvokliknite na nju i dobivenoj funkciji dodajte sljedeći kôd:

```
private void menuIzlaz_Click(object sender, System.EventArgs e)
{
   Close();
}
```

Ostale stavke u ovoj skupini ostavljamo za kasnije, a sada ćemo za pozabaviti stavkama u skupini Uređivanje. Za prvih pet stavki kôd dajemo u kompletu, a koji pripada kojoj otkrit ćete prema imenima funkcija:

```
private void menuUndo_Click(...)
{
   textBox1.Undo();
}

private void menuKopiraj_Click(...)
{
   textBox1.Copy();
}

private void menuIzrezi_Click(...)
{
   textBox1.Cut();
}

private void menuZalijepi_Click(...)
{
   textBox1.Paste();
}

private void menuOznaciSve_Click(...)
{
   textBox1.SelectAll();
}
```

Jednostavno, zar ne? Dobro, moramo priznati, kao primjere smo koristili upravo one opcije za koje postoje gotove metode, dok smo lukavo izbjegli one koje zahtijevaju ikakvo dodatno programiranje. Međutim, niti takvi slučajevi nisu nuklearna fizika – često se stvari svode na tek nekoliko linija, kao što ćemo vrlo brzo vidjeti u primjerima snimanja i učitavanja datoteka.

No, prije toga, pozabavimo se stavkom WordWrap, koja će imati ulogu uključivanja i isključivanja svojstva WordWrap koje pripada polju za unos teksta. Kod ovakvih stavaka vrlo je važno da prije svega uskladite početne vrijednosti svojstva Checked na stavci i svojstva WordWrap na polju za unos. Drugim riječima, ukoliko ste u dizajnerskom načinu svojstvo WordWrap ostavili uključeno, onda morate uključiti i svojstvo Checked.

Ovakvo ručno podešavanje nije problem u jednostavnim primjerima poput ovoga, no može stvoriti probleme u složenijim aplikacijama. Zašto onda ne bismo brigu o tome prepustili računalu? Napravit ćemo da se prilikom pokretanja aplikacije provjerava stanje svojstva WordWrap i sukladno tome postavlja vrijednost svojstvu Checked. Kôd je sljedeći:

```
menuWordWrap.Checked = textBox1.WordWrap;
```

Ali, gdje ga staviti? Mogućnosti ima puno i sve će u našem primjeru bez problema raditi, no to ne mora vrijediti za neke druge složenije slučajeve. Jedno od rješenja je smjestiti kôd u konstruktor Form1(), na mjesto gdje je automatski generiran komentar "TODO: Add any constructor code after InitializeComponent call". Kako su konstruktori metode koje se izvršavaju prilikom stvaranja objekta, možemo biti sigurni da će se vrijednosti svojstava uskladiti na samom početku.

Osim tamo, kôd možete smjestiti i u funkciju vezanu uz događaj forme Load koji nastupa prilikom učitavanja forme, a ukoliko je moguće da neko svojstvo bude promijenjeno na više mjesta u programu, najsigurnije je raditi usklađivanje neposredno prije nego što se roditeljska stavka otvori. U tom slučaju kôd ćemo upisati u funkciju vezanu uz događaj PopUp, no ne na stavci na kojem ga mijenjamo nego na roditeljskoj stavci kojoj ona pripada. U našem slučaju, vezali bismo se na događaj PopUp objekta menuUredjivanje.

Pretpostavljamo da ste iz količine teksta koju smo posvetili usklađivanju shvatili koliko je ono važno. Takvi, naizgled sitni propusti, vrlo snažno utječu na korisnika zbunjujući ga i smanjujući povjerenje u program i autora. Jedna od glavnih snaga svih iole popularnijih aplikacija je dorađenost sitnica.

Evo konačno i funkcije koju valja vezati uz klik na stavku WordWrap:

```
private void menuWordWrap_Click(...)
{
    // inverzija svojstva WordWrap:
    textBox1.WordWrap = (!(textBox1.WordWrap));
```

```
    // usklađivanje sa stavkom u izborniku:
    menuWordWrap.Checked = textBox1.WordWrap;
    // prethodna linija nije potrebna ako radimo
    // usklađivanje uz događaj PopUp
}
```

Rad s datotekama

Kad u naprednim uređivačima teksta odaberete opciju otvaranja novog dokumenta, otvorit će vam se nov, prazan list papira u novom prozoru. U jednostavnijim editorima (kakav je naš), koji nemaju mogućnost otvaranja više dokumenata istovremeno, otvaranje novog dokumenta jednako je brisanju sadržaja postojećeg polja za upis. Stoga nam je funkciju otvaranja novog dokumenta vrlo jednostavno napisati:

```
private void menuNovi_Click(...)
{
    textBox1.Text = "";
}
```

Dijaloški okvir za otvaranje datoteka

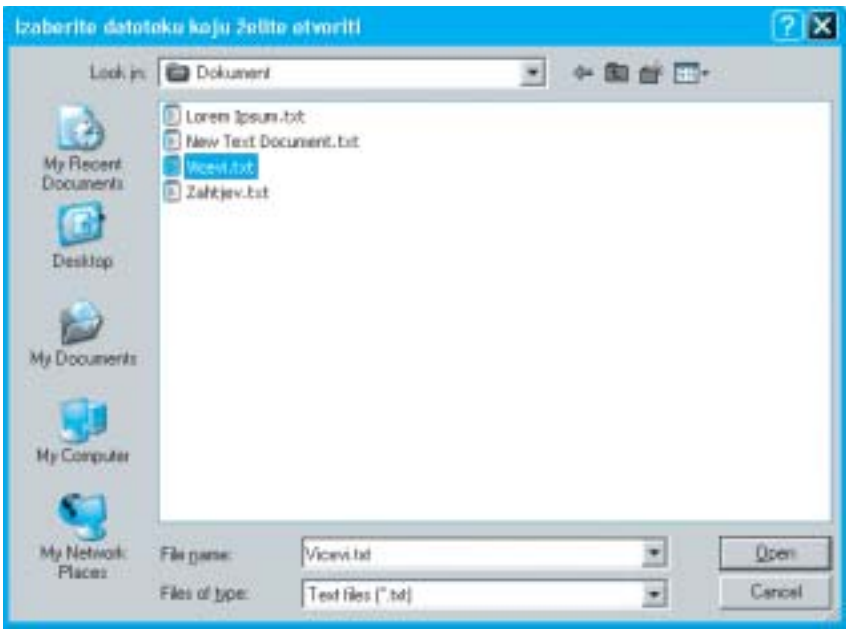
Nešto složenija funkcionalnost je učitavanje datoteke s diska i prikaz njezina sadržaja u polju za unos teksta. Taj ćemo posao podijeliti u dva dijela. Prvi nam je zadatak ponuditi korisniku popis datoteka na disku i mogućnost da jednu od njih izabere, dok će nam drugi biti otvaranje izabrane datoteke i kopiranje njezina sadržaja u textBox1.

Prvi zadatak, iako se na prvi pogled čini složen, zapravo je vrlo jednostavan. Naime, za tu ćemo funkcionalnost koristiti kontrolu OpenFileDialog koji nam sve potrebno donosi kao na pladnju. Dodajmo, stoga, kontrolu OpenFileDialog na formu (odnosno pokraj nje) i pozabavimo se nekim njezinim svojstvima.

OpenFileDialog zapravo je dijaloška forma koja na sebi ima sve potrebne kontrole koje omogućavaju korisniku da odabere datoteku. Prije nego što je prikažemo na ekranu, zgodno je podesiti svojstva koja utječu na njen izgled i ponašanje. Tako, primjerice, svojstvo Title određuje koji će se naslov naći u zaglavlju prozora.

Svojstvo InitialDirectory određuje mapu koja će biti prikazana prilikom otvaranja prozora. S ovim svojstvom treba postupati oprezno jer nema stopostotne garancije da mapa koju ovdje navedete postoji na računalu osobe koja koristi program. Zato u njega valja trpati vrijednosti isključivo na temelju prethodnog odabira mape, analize korisnikovog diska ili slično. U slučaju da, kao mi u

primjeru, ne upišete nikakvu vrijednost, program će kao inicijalnu mapu otvarati "My Documents" odnosno zadnju korištenu mapu.



Slika 8-18: ***Dijaloška forma OpenFileDialog***

Svojstvo RestoreDirectory jedno je od onih koji vam može natjerati suze na oči ako na njega zaboravite. Naime, inicijalno je isključeno, a to znači da će svaki put kada otvorite datoteku pomoću dijaloškog okvira OpenFileDialog mapa u kojoj se datoteka nalazi postati radna mapa programa. Zato preporučujemo da vrijednost ovog svojstva postavite na *true* kako vam ne bi interferirao s radnom mapom.

Radna mapa aplikacije je, ako nije drugačije postavljeno, ona mapa u kojoj se nalazi i izvršna datoteka aplikacije. Radnu mapu možete mijenjati u prečici programa (desni klik na *shortcut*, Properties, polje "Start in") ili pak u samom programu. Putanju radne mape u svakom trenutku možete pročitati iz izraza Environment.CurrentDirectory. Inače, radna mapa služi kao mjesto gdje aplikacija traži sve datoteke kojima nije navedena putanja. Drugim riječima, ukoliko u kodu na neku datoteku referencirate bez njezine putanje, podrazumijevat će se da se nalazi u radnoj mapi.

Vjerojatno najvažnije svojstvo ovog dijaloškog okvira je ono koje je nazvano Filter. Pomoću njega određujemo sadržaj padajuće liste "Files of type" (vidi dno slike 8-18) odnosno omogućavamo filtriranje prikazanih datoteka prema njihovoj ekstenziji. To svojstvo, u usporedbi s drugima, ima vrlo neobičnu sintaksu, a kod pisanja iste morat ćete se osloniti isključivo na sebe jer ne postoji nikakav pomoćni prozor.

Radi se po posebnim pravilima napisanom *stringu*. Primjerice, želite li omogućiti korisnicima učitavanje tekstualnih datoteka (dakle, ekstenzija će biti "txt"), svojstvo Filter glasit će ovako:

```
Text files (*.txt)|*.txt
```

U prvi dio možete upisati što vam je drago (iako je način iz primjera uvriježen i ne preporučujemo izmišljanje tople vode), dok u drugom dijelu, nakon znaka "|", upisujete sam filtar (zvjezdica zamjenjuje neodređen broj bilo kojih znakova).

Ako želite da padajuća lista "Files of type" ima više članova, napisat ćete nešto poput ovoga, po formuli opis, crta, filtar, crta, opis, crta, filtar...

```
Text files (*.txt)|*.txt|All files (*.*)|*.*
```

Jednom članu padajuće liste možete dodijeliti i više ekstenzija. To onda izgleda ovako:

```
Images (*.jpg; *.gif; *.png)|*.jpg;*.gif;*.png
```

Svojstvom FilterIndex određujete koji će od članova padajuće liste inicijalno biti odabran (obavezno obratite pažnju na upozorenje uz tekst!).

Za razliku od drugih indeksa, prvi član u svojstvu FilterIndex određuje se brojkom 1 (a ne 0, kako smo navikli).

Svojstva CheckFileExists i CheckPathExists gotovo sigurno nećete mijenjati kod dijaloškog okvira za otvaranje datoteka jer je sasvim logično da datoteku možete otvoriti jedino ako postoji, a ova svojstva, ako su uključena, provjeravaju upravo to. Još manje utjecaja imaju svojstva AddExtension i DefaultExt koja ćemo obraditi malo kasnije, kod dijaloškog okvira za snimanje.

U svojstvo FileName zapisuje se ime i putanja do datoteke koju je korisnik odabrao i ona se u pravilu ne koristi prije, nego nakon pozivanja dijaloškog okvira.

U slučaju da želite otvoriti više datoteka odjednom (što u našem primjeru nema smisla, no može vam zatrebati), prije poziva okvira svojstvu MultiSelect dodijelite vrijednost *true*. Imajte na umu da ćete u tom slučaju sami iz parametra FileName morati razlučiti putanje svih odabranih datoteka.

Pretpostavljamo da svojstva podešavate kroz pomoćni prozor sučelja Visual Studija, no pozivanje dijaloškog okvira možete napraviti samo iz kôda, najčešće vezano uz neki događaj. Kod nas će to biti funkcija vezana za odabir stavke Otvori u izborniku. Pozivanje okvira radimo metodom ShowDialog(), a ona se gotovo uvijek koristi u ovakvom obliku:

```
private void menuOtvori_Click(...)
{
   if (openFileDialog1.ShowDialog() == DialogResult.OK)
   {
      // kôd za čitanje datoteke
   }
}
```

Koncentrirajmo se za trenutak na liniju koja počinje naredbom *if*. U uvjetu koji nakon naredbe koristimo primjećujete kako metodu za pozivanje dijaloškog okvira uspoređujemo s vrijednošću DialogResult.OK. Naime, ta metoda vraća vrijednost tipa DialogResult. Ukoliko je korisnik u okviru izabrao datoteku i pritisnuo gumb OK, onda će vraćena vrijednost biti jednaka DialogResult.OK. Ukoliko je odustao pritisnuvši gumb Cancel, onda će vrijednost biti DialogResult.Cancel. Slijedi sasvim logičan zaključak – kôd koji će odraditi otvaranje datoteke treba biti izvršen samo ako je korisnik kliknuo OK. Zato vraćenu vrijednost metode ShowDialog() uspoređujemo s adekvatnom vrijednošću tipa DialogResult i samo ako su identične dozvoljavamo izvršavanje kôda za otvaranje datoteke.

Spomenuti način provjere, kao što ćemo vidjeti, jednak je za sve dijaloške okvire, s tim da neki, primjerice, vraćaju vrijednosti DialogResult.Yes i DialogResult.No.

Čitanje datoteke s diska

Dopunimo prošli komad kôda i dodajmo unutar bloka pod paskom naredbe *if* četiri linije za otvaranje datoteke i čitanje njezina sadržaja:

```
private void menuOtvori_Click(...)
{
   if (openFileDialog1.ShowDialog() == DialogResult.OK)
```

```
    {
        string ImeDatoteke = openFileDialog1.FileName;
        System.IO.StreamReader Datoteka
                = new System.IO.StreamReader(ImeDatoteke);
        textBox1.Text = Datoteka.ReadToEnd();
        Datoteka.Close();
    }
}
```

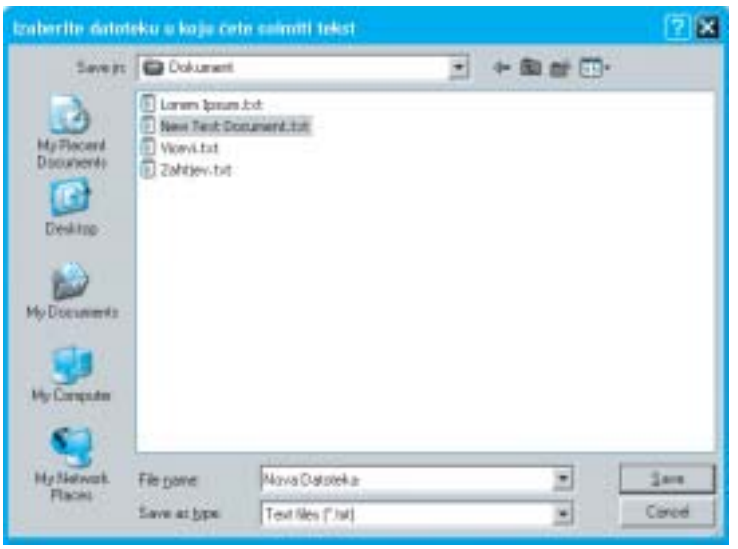
U prvoj liniji deklariramo varijablu ImeDatoteke tipa *string,* kojoj odmah pridružujemo ime i putanju datoteke koja je izabrana pomoću dijaloškog okvira (svojstvo FileName).

Sljedeći redak stvara objekt nazvan Datoteka. To je objekt tipa StreamReader (u klasi System.IO) koji, između ostalog, omogućava čitanje datoteka s diska. Kao parametar konstruktora navodimo varijablu ImeDatoteke koja sadrži ime i putanju do datoteke. Svaki put kada pristupamo tom objektu zapravo radimo s datotekom koju je korisnik odabrao.

Treća linija bloka u svojstvo textBox1.Text sprema sadržaj datoteke koji uzimamo metodom ReadToEnd(). U zadnjoj liniji zatvaramo datoteku jer je više nećemo koristiti.

Dijaloški okvir za snimanje datoteka

Za snimanje podatka u datoteku predvidjeli smo dvije stavke: Snimi (*Save*) i Snimi kao (Save as). Dijaloški okvir za snimanje datoteka trebat će nam prvenstveno u drugoj, iako se potreba za njime može pojaviti i u prvoj ukoliko datoteka još nije bila snimana.



Slika 8-19: ***Dijaloška forma SaveFileDialog***

Funkcija vezana uz događaj odabira stavke Snimi kao izgledat će ovako:

```
private void menuSnimiKao_Click(...)
{
   if (saveFileDialog1.ShowDialog() == DialogResult.OK)
   {
      // kôd za snimanje datoteke
   }
}
```

Uočite da je pozivanje ovog dijaloškog okvira identično onome za otvaranje datoteka, čak se koristi i isti tip vraćene vrijednosti. Osim toga, ove dvije kontrole dijele i neka zajednička svojstva tako da dio priče o svojstvima kontrole OpenFileDialog vrijedi i ovdje.

Međutim, dio tih svojstava u ovom slučaju ima više smisla. Tu prvenstveno mislimo na svojstva CheckPathExists i CheckFileExists. Kod snimanja je, kao i kod otvaranja, bitno da putanja (CheckPathExists) postoji jer u protivnom, ako korisnik ručno upiše ime mape koja ne postoji, datoteka u njoj neće moći biti kreirana (u slučaju da to dozvolite, prije snimanja ćete u kodu morati kreirati tu mapu). Međutim, svojstvo CheckFileExists bi trebalo biti na vrijednosti *false* jer kod snimanja datoteke ne samo da nije nužno da datoteka već postoji nego je i poželjno da ne bude tako. To ne znači da korisnik neće moći prepisati neku postojeću datoteku, no to je već domena svojstva OverwritePrompt koji, ako je uključen, upozorava korisnika da će prepisati postojeću datoteku ukoliko upiše ime datoteke koje već postoji.

Ako iz nekog razloga želite da korisnik bude upozoren i ako kreira sasvim novu datoteku, uključit ćete svojstvo CreatePrompt. Ova se mogućnost koristi vrlo rijetko.

Svojstva AddExtension i DefaultExt također ovdje imaju puno više smisla, iako smo ih susretali i kod prošlog dijaloškog tipa. Naime, korisnici su navikli ne pisati ekstenzije svojim datotekama, već očekuju da se o tome brine sustav. Primjerice, u programima za editiranje tekstualnih datoteka korisnik logično je da će se automatski dodati ekstenzija "txt", pa smo mi u svom primjeru svojstvo AddExtension postavili na *true*, a u svojstvo DefaultExt zapisali ekstenziju koju želimo da bude dodana, ako je korisnik sam ne upiše (dakle, upisali smo vrijednost "txt"). Kao rezultat toga, kada korisnik upiše samo "datoteka", dijaloški okvir će automatski dodati ekstenziju i u svojstvu FileName vratiti vrijednost "c:\neka putanja\datoteka.txt".

Snimanje datoteka na disk

Sljedeći zadatak nam je napisati kôd koji će snimati sadržaj tekstualnog polja u datoteku. Kako ćemo tu funkcionalnost koristiti na dva mjesta (kod stavaka Snimi i Snimi kao), preporučljivo je da napravimo zasebnu funkciju koja će imati tu funkcionalnost te je kasnije pozivamo s mjesta na kojem nam zatreba:

```
private void snimanjeDatoteke(string ImeDatoteke)
{
    System.IO.StreamWriter Datoteka
        = new System.IO.StreamWriter(ImeDatoteke, false);
    Datoteka.Write(textBox1.Text);
    Datoteka.Close();
}
```

Kao parametar ove funkcije tražimo ime datoteke i zapisujemo u varijablu ImeDatoteke te je kasnije u kodu koristimo. Primijetit ćete da u ovom primjeru koristimo drugi tip objekta (System.IO.-StreamWriter) koji služi za pisanje u datoteke. Prilikom kreiranja objekta opet navodimo ime datoteke, a kao drugi parametar pojavljuje se opcija dodavanja ili prepisivanja datoteke. Konkretno, ako kao drugi parametar navedemo vrijednost *false,* stara će datoteka biti prepisana, a ako stavimo vrijednost *true,* onda će ono što pišemo biti dodano na kraj postojeće datoteke. Naravno, sve pod uvjetom da stara datoteka istog imena postoji – ukoliko ne postoji, parametar nema utjecaja na ponašanje objekta.

Ostale dvije linije sigurno i sami razumijete. Prva metodom Write() u datoteku zapisuje sadržaj svojstva textBox1.Text, a druga zatvara datoteku.

Još nam preostaje napisati kôd za stavke Snimi i Snimi kao. Počet ćemo s potonjom:

```
private void menuSnimiKao_Click(...)
{
    if (saveFileDialog1.ShowDialog() == DialogResult.OK)
    {
        snimanjeDatoteke(saveFileDialog1.FileName);
    }
}
```

Kako već imamo pripremljenu funkciju za snimanje datoteke, pozivamo je, a kao parametar joj navodimo ime i putanju datoteke koju je korisnik izabrao pomoću dijaloškog okvira za snimanje.

Stavka Snimi nešto je složenija. Razmislimo malo... Ukoliko dokument još nije bio sniman, onda se stavka Snimi treba ponašati isto kao i stavka Snimi kao – otvoriti dijaloški okvir i pitati za ime. Međutim, ukoliko je dokument bio sniman ili smo otvorili postojeću datoteku, onda znamo njezino ime i nema potrebe da program otvara dijaloški okvir za biranje imena.

Stoga nam treba mjesto gdje ćemo zapisati ime datoteke kada je otvorimo i/ili snimimo. Slobodni smo za to stvoriti na nivou klase posebnu varijablu tipa *string*, no puno zgodnije rješenje je iskorištavanje jednog dosad nespomenutog svojstva, prisutnog kod apsolutno svih kontrola. Svojstvo se zove Tag i pripada tipu *object,* što znači da u njega možemo utrpati bilo što, uključujući i ime datoteke. Nadogradimo malo naše funkcije kako bi ime datoteke zapisali u svojstvo Tag kontrole textBox1:

```
private void menuNovi_Click(...)
{
   textBox1.Text = "";
   textBox1.Tag = "";
}

private void menuOtvori_Click(...)
{
   if (openFileDialog1.ShowDialog() == DialogResult.OK)
   {
      string ImeDatoteke = openFileDialog1.FileName;
      System.IO.StreamReader Datoteka
               = new System.IO.StreamReader(ImeDatoteke);
      textBox1.Text = Datoteka.ReadToEnd();
      Datoteka.Close();
      textBox1.Tag = ImeDatoteke;
   }
}

private void snimanjeDatoteke(string ImeDatoteke)
{
   System.IO.StreamWriter Datoteka
      = new System.IO.StreamWriter(ImeDatoteke, false);
   Datoteka.Write(textBox1.Text);
   Datoteka.Close();
   textBox1.Tag = ImeDatoteke;
}
```

U slučaju stvaranja novog dokumenta on neće automatski imati pripadajuće ime datoteke, pa svojstvu textBox1.Tag pridružujemo prazan *string*. Kod učitavanja datoteke, u to svojstvo zapisujemo ime datoteke koje je korisnik izabrao u dijaloškom okviru, a kod snimanja koristimo parametar funkcije snimanjeDatoteke u kojem se ime datoteke nalazi. Promjena imena datoteke vezane uz dokument koji editiramo može se, dakle, dogoditi u tri slučaja – prilikom stvaranja novog dokumenta, otvaranja druge datoteke s diska i snimanja datoteke.

Nakon što smo se uvjerili da se u svojstvu textBox1.Tag uvijek nalazi pravo ime datoteke i da će to svojstvo, ukoliko ime ne postoji, biti prazno, možemo napisati funkciju za stavku Snimi:

```
private void menuSnimi_Click(object sender, System.EventArgs e)
{
   if (textBox1.Tag.ToString() != "")
```

```
        {
            snimanjeDatoteke(textBox1.Tag.ToString());
        }
        else
        {
            if (saveFileDialog1.ShowDialog() == DialogResult.OK)
            {
                snimanjeDatoteke(saveFileDialog1.FileName);
            }
        }
    }
```

Kako je textBox1.Tag tipa *object*, bitno je pomoću metode ToString() napraviti konverziju u *string,* kako bismo to svojstvo mogli koristiti u uvjetu. Ukoliko ime datoteke postoji (svojstvo Tag nije prazan *string*), poziva se funkcija za snimanje datoteke, a kao parametar navodimo ime datoteke odnosno isto to svojstvo Tag. U suprotnom slučaju, izvršit će se izraz identičan onome u funkciji vezanoj uz stavku Snimi kao. Umjesto te tri linije, mogli smo napisati i sljedeće:

```
menuSnimiKao.PerformClick();
```

Metoda PerformClick(), naime, simulira klik na objektu kojem pripada, nastupa događaj Click te se izvršava funkcija vezana uz njega (u našem slučaju, funkcija menuSnimiKao_Click).

Snimanje umjesto gubitka teksta

Ukoliko radite po primjerima i isprobavate sve što radimo (a to toplo preporučujemo), primijetit ćete da nam nedostaje još jedna sitnica kako bi se naša aplikacija ponašala u skladu s običajima. Naime, naučili smo da prilikom stvaranja novog dokumenta ili otvaranja nekog drugog, program upozorava da postojeći dokument nije snimljen i ponudi njegovo snimanje. U našem primjeru takvog ponašanja nema pa ako, primjerice, slučajno kliknete na novi dokument, a postojeći niste snimili, sve će vam promjene otići u nepovrat.

Zato nam treba pokazatelj koji će u svakom trenutku znati je li tekst u polju za upis promijenjen od zadnjeg učitavanja, snimanja odnosno stvaranja novog dokumenta.

Da bismo riješili taj problem, za početak valja definirati varijablu tipa *boolean* na nivou cijele klase. To radimo tako da na vrhu kôda, u predjelu za deklaraciju varijabli, iznad konstruktora klase public Form1(), upišemo sljedeće:

```
private bool Mijenjano = false;
```

Na taj način početnu vrijednost varijable Mijenjano stavljamo na *false*. U slučaju da se tekst u kontroli textBox1 izmijeni, varijablu bi trebalo postaviti na *true*. Zato sljedeću funkciju vežemo uz događaj TextChanged:

```
private void textBox1_TextChanged(...)
{
   Mijenjano = true;
}
```

Zahvaljujući ovom kodu, vrijednost varijable Mijenjano bit će gotovo uvijek *true*. Ipak, ne zaboravimo da ova varijabla označava je li tekst mijenjan od zadnjeg snimanja. Sukladno tome, neposredno nakon snimanja varijabla Mijenjano bi trebala imati vrijednost *false*. Ista stvar je i u situacijama neposredno nakon otvaranja dokumenta (tekst je identičan onome u datoteci, što znači da nije mijenjan) odnosno nakon stvaranja novog dokumenta (dokument je prazan, pa se nema što snimati). Zato u sve te funkcije (a to su, igrom slučaja, sve one u koje smo dodavali liniju za pamćenje naziva datoteke) dodajemo sljedeću liniju kôda:

```
Mijenjano = false;
```

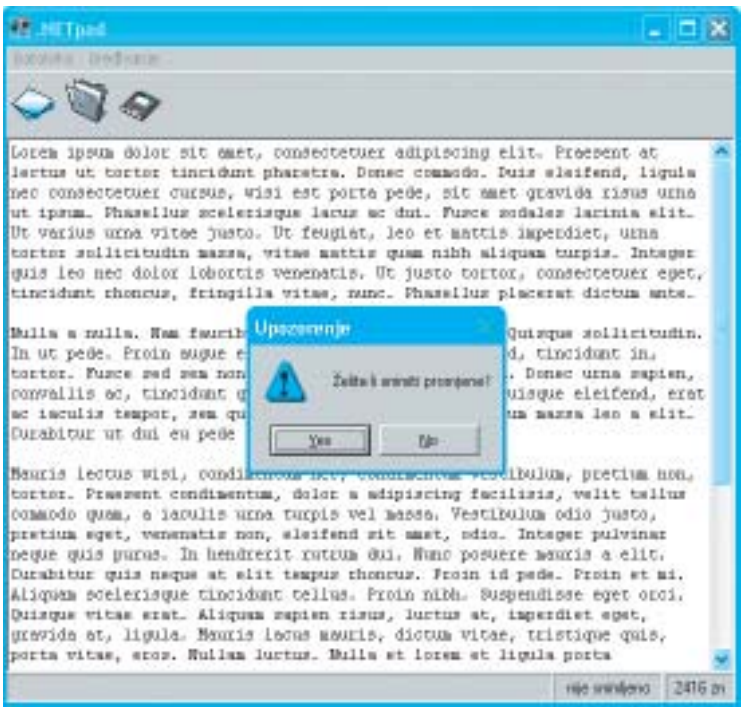
Pokazatelja je li tekst promijenjen imamo, pa nam preostaje samo njegovo korištenje. Zato ćemo napisati jednostavnu funkciju:

```
private void provjeraSnimljenosti()
{
   if (Mijenjano)
   {
      if (MessageBox.Show(
              "Želite li snimiti promjene?",
              "Upozorenje",
              MessageBoxButtons.YesNo,
              MessageBoxIcon.Exclamation) == DialogResult.Yes)
      {
         menuSnimi.PerformClick();
      }
   }
}
```

Osnovna zadaća ove funkcije je da, ukoliko je tekst mijenjan, pozove dijaloški okvir s pitanjem "Želite li snimiti promjene?". Ako korisnik odgovori potvrdno, pokrenut će se procedura snimanja simulacijom klika na izborničku stavku Snimi. Ona je u ovom slučaju idealna – ukoliko postoji ime dokumenta, snimit će promjene pod tim imenom, a ukoliko ne postoji, otvorit će dijaloški okvir za upis imena.

Dio u primjeru koji još nismo sreli odnosi se na dijaloški okvir MessageBox. Iako se na prvi pogled čini slična dijaloškim okvirima koje smo upoznali, ona ima specifičnu karakteristiku da nije po-

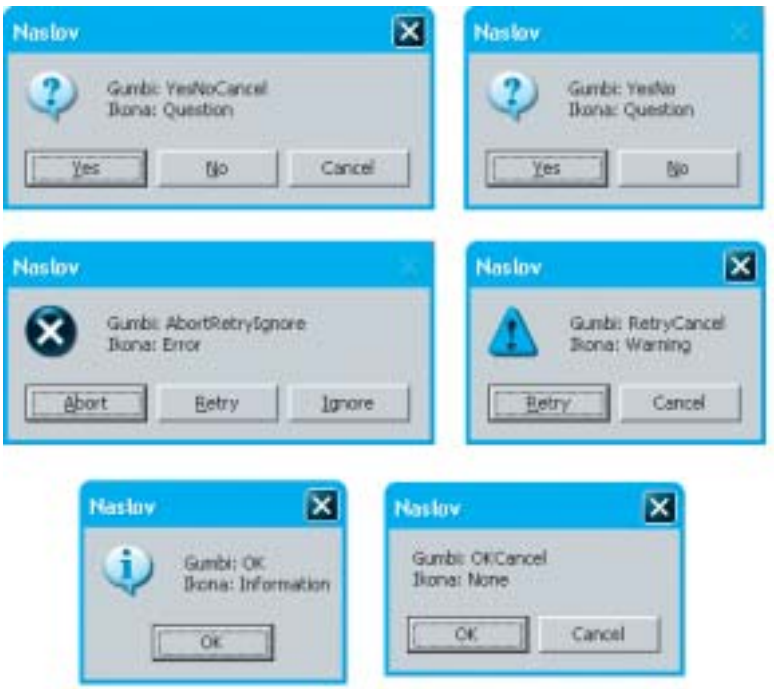
trebno (zapravo, nije moguće) napraviti novu instancu te klase odvlačenjem pripadajuće kontrole na formu. Ona se koristi bez ikakvih predradnji, jednostavnim pozivanjem statične (zajedničke; vidi prethodno poglavlje) metode Show() kojom kao parametre navodimo svojstva koja želimo da ima dijaloški okvir.



Slika 8-20: ***Želite li snimiti promjene, pita MessageBox.***

Metoda Show() je preopterećena što znači da može imati različite "komplete" parametara, a mi smo u primjeru izabrali jedan, u ovom slučaju najpraktičniji među njima. On ima četiri parametra – prvi predstavlja tekst (u našem slučaju pitanje) koji će se pojaviti, drugi služi za definiranje naslova u zaglavlju, treći određuje koji će gumbi biti dostupni (u našem slučaju Yes i No), a četvrti ikonu koja će biti prikazana (mi smo odabrali uskličnik). Na slici 8-21 možete vidjeti razne kombinacije spomenutih parametara.

Želite li u tekstu MessageBox-a prijeći u novi redak, koristite izraz "\n" (bez navodnika).



Slika 8-21: ***Gumbi i ikone dostupni u MessageBox-u (neke od dostupnih ikona nisu prikazane, jer predstavljaju stari način imenovanja)***

Koji je gumb korisnik pritisnuo otkriva se na isti način kao i kod ostalih dijaloških okvira. Naravno, valja paziti na povezanost vrijednosti koje mogu biti vraćene i prikazanih gumbi, posebno stoga što vas kompajler o propustima tog tipa neće obavijestiti.

Konačno, napisanu funkciju treba pozvati prije svih radnji koje mogu rezultirati gubitkom podataka u kontroli textBox1. Tri su takve radnje – stvaranje novog dokumenta, otvaranje drugog dokumenta i izlaz iz programa. Stoga poziv te funkcije treba smjestiti na sam početak tih funkcija. Evo kako će to izgledati:

```
private void menuNovi_Click(...)
{
   provjeraSnimljenosti();
   textBox1.Text = "";
   textBox1.Tag = "";
}

private void menuOtvori_Click(object sender, System.EventArgs e)
{
   provjeraSnimljenosti();
```

```
    if (openFileDialog1.ShowDialog() == DialogResult.OK)
    {
        // dio kôda je izostavljen
    }
}
```

Kod izlaza iz programa ovaj način ne "pali" jer je, osim klikom na stavku Izlaz, iz programa moguće izaći klikom na crveni križić u gornjem desnom uglu ili kombinacijom tipaka Alt + F4. Zato tu funkciju treba vezati uz događaj forme, i to onaj koji nastupa neposredno prije zatvaranja – Closing. Sâm kôd je vrlo jednostavan:

```
private void Form1_Closing(...)
{
    provjeraSnimljenosti();
}
```

Traka s alatima

Kontrola ToolBox služi za kreiranje trake s alatima. Takav izbor najčešće korištenih funkcija možemo susresti u gotovo svim aplikacijama.

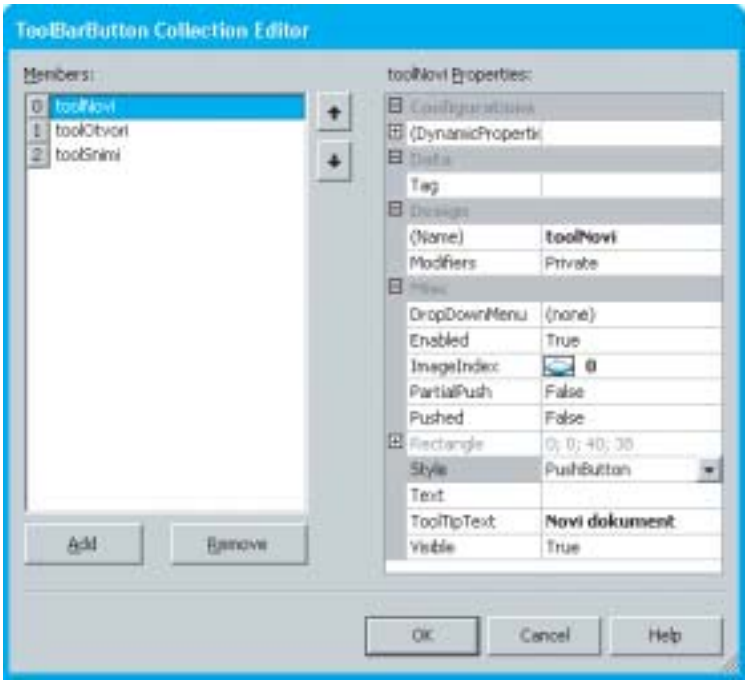
Prvi korak je postavljanje kontrole na formu, nakon čega će se ona automatski prilijepiti uz gornji obod forme. Nakon toga joj treba definirati pravila izgleda i ponašanja te naposljetku dodati određeni broj gumbi i odrediti im funkcionalnosti. Krenimo redom...

Među vizualnim svojstvima trake s alatima pronaći ćemo brojne već poznate stavke, ali i neke nove, specifične za ovu kontrolu. Od poznatih spominjemo tek BorderStyle koji određuje tip obruba, dok od novih valja spomenuti svojstvo Divider koji određuje postojanje linije koja razdvaja sam vrh forme (odnosno izbornik, ako postoji) s alatnom trakom. Svojstvo Wrappable određuje hoće li gumbi na traci moći biti prikazani i u više redova ukoliko veličina prozora to zahtijeva.

Gumbi na traci

Dodavanje gumba na alatnu traku rješava se preko svojstva Buttons. Klik na gumbić kraj tog svojstva otvorit će pomoćni prozor (slika 8-22) unutar kojeg ćemo dodavati gumbe i dodjeljivati im svojstva.

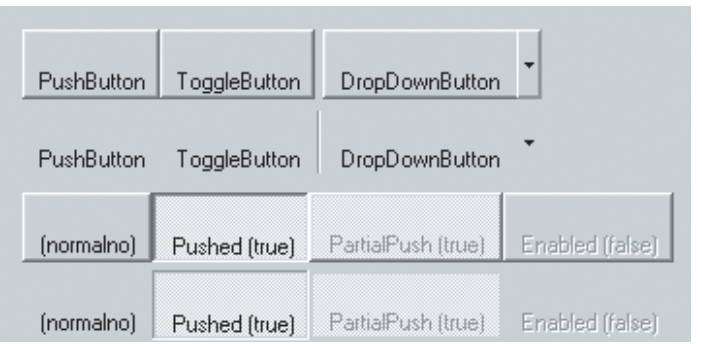
Gumbi na alatnoj traci također su kontrole (tipa ToolBarButton), no njihovo dodavanje u Visual Studijevom sučelju riješeno je na drugačiji način nego što je to učinjeno sa stavkama izbornika koje su u vrlo sličnom odnosu sa svojom roditeljskom kontrolom (MainMenu). Ipak, na ovo ne možemo uzeti dokaz nedosljednosti jer je izbornik značajno složeniji od alatne trake pa zahtjeva i drugačije rješenje. Naime, svaka (pod)stavka može imati vlastite podstavke, dok gumbi alatne trake mogu biti isključivo u istom nivou.



Slika 8-22:
Pomoćni prozor za dodavanje gumba

Kako je svaki gumb zapravo kontrola (vidi okvir), svaki od njih će imati zasebna svojstva koja možemo prilagođavati na desnoj strani pomoćnog prozora. Ovdje vrijedi isti savjet kao i za stavke izbornika – imenujte ih logično kako biste se kasnije lakše snalazili.

Slika 8-23:
Različiti tipovi, stilovi i pojave gumba



Gumbi se na alatnoj traci mogu pojavljivati u dva oblika – kao klasični izbočeni gumbi ili moderniji ravni gumbi. To odlučujemo na nivou cijele alatne trake mijenjajući njeno svojstvo Appearance. Drugim riječima, nije moguće na istoj traci imati i jedan i drugi tip gumba.

Kako će se gumb ponašati određujemo za svaki posebno (dakle, u pomoćnom prozoru iza svojstva Buttons). Tamo možemo odabrati stil gumba koji može biti klasičan (PushButton), sklopni (ToggleButton), odjelni (Separator) i s padajućim izbornikom (DropDownButton). Kako koji stil izgleda u praksi možete vidjeti na slici 8-23 (prvi red su izbočeni, a drugi ravni).

Na toj slici postoje dvije stvari koje vas mogu zbuniti. Prvo je razlika između stila PushButton i ToggleButton – vizualno, među njima nema razlike, no pokrenete li aplikaciju, uočit ćete da se drugačije ponašaju. Naime, kad kliknete na gumb sa stilom PushButton, on će se nakon klika vratiti u početni položaj, dok će onaj sa stilom ToggleButton ostati pritisnut sve dok ga ponovo ne pritisnete. Osim vizualnog ponašanja, gumbima različitih stilova ćete drugačije i postupati u kodu – PushButton će izvršavati određenu radnju dok će ToggleButton uključivati odnosno isključivati određenu opciju, baš kao da se radi o stavkama u izborniku među kojima jedna ima, a druga nema kvačicu.

Druga zbunjujuća okolnost vezana uz sliku 8-23 jest gumb stila Separator. Naime, to zapravo i nije pravi gumb već linija koja odvaja dvije skupine gumba, opet vizualno na isti način kao što smo to radili u izbornicima. Međutim, tu već možemo pričati o nedosljednosti – dok u izbornicima separator dodajemo upisujući znak minusa umjesto teksta, ovdje to radimo pomoću svojstva stila.

Slika 8-24:
Gumb s padajućim izbornikom u akciji

Kod gumba stila ToggleButton postoji svojstvo preko kojeg možemo saznati je li gumb u pritisnutom stanju. Radi se o svojstvu Pushed koje poprima *boolean* vrijednosti. Osim njega, tu je i svojstvo

PartialPush koje, ako je uključeno, na neobičan način zasivljuje gumb iako su sve njegove funkcije normalne. Kako to izgleda u praksi, možete vidjeti u zadnja dva retka slike 8-23, a osim spomenutih svojstva prikazali smo i svojstvo Enabled. Ono je moguće uključiti gumbima svih stilova, a osim što ih zasivljuje, ujedno i onemogućava njihovo korištenje.

Stil DropDownButton omogućava da gumbu dodamo i padajući izbornik. Padajući izbornik koji će se pojaviti, pogađate, posebna je kontrola. Radi se o kontroli tipa ContextMenu, koju je potrebno zasebno dovući na formu i povezati je s gumbom preko svojstva DropDownMenu.

U padajućoj listi svojstva DropDownMenu pronaći ćete i izbornik tipa MainMenu, kao sve izborničke stavke koje postoje na formi, no jedini dozvoljeni izbor je kontrola tipa ContextMenu.

ContextMenu je vrlo sličan kontroli MainMenu, s tom razlikom da se sve stavke moraju smjestiti ispod jedne, glavne "stavke" nazvane ContextMenu. Ta će se glavna "stavka" prikazati samo u dizajnerskom načinu rada.

Kontrolu ContextMenu možete vezati uz velik broj kontrola. Primjerice, ako je vežete uz formu preko svojstva ContextMenu, onda će taj izbornik biti prikazan kada korisnik na formu klikne desnom tipkom miša. Isto vrijedi i za brojne druge kontrole.

Gumbi stila DropDownButton mogu svoj padajući izbornik pokazati na dva načina. Prvi, uobičajeniji, kakav možete vidjeti na slici 8-24, uključuje postojanje strelice pritiskom na koju se izbornik otvara. U tom slučaju gumb može imati jednu funkcionalnost, a kroz stavke padajućeg izbornika mogu se ponuditi dodatne, u praksi srodne funkcionalnosti. Ako isključimo postojanje strelice (svojstvo DropDownArrows, na nivou cijele alatne trake), padajući izbornik će se otvarati klikom na bilo koji dio gumba i on neće moći imati vlastitu funkcionalnost prilikom tog klika (drugim riječima, klikom na takav gumb neće nastupiti događaj ButtonClick, koji ćemo uskoro upoznati).

U Visual Studiju .NET 2003 postoji greška vezana uz svojstvo DropDownButton. Naime, dokumentacija govori da je defaultna vrijednost *false* i to funkcionira u dizajnerskom načinu, no kad pokrenete aplikaciju primijetit ćete da je svojstvo uključeno. Kada želite da to bude tako, vrijednost svojstva stavite na true i stvar će funkcionirati, no ako želite isključiti postojanje strelice, morat ćete se poslužiti trikom. U konstruktor forme na kojoj se alatna traka nalazi ubacite sljedeći kôd:

```
toolBar1.DropDownArrows = false;
```

Svaki gumb u alatnoj traci može imati tri karakteristike koje će korisniku govoriti o njegovoj funkcionalnosti. Tekst, sliku (ili, prikladnije rečeno, ikonu) i tzv. *tooltip*, mali žuti pravokutnik koji se pojavi s objašnjenjem nakon što nekoliko trenutaka zadržite miša iznad gumba (ili neke druge kontrole).

Tekst svakog gumba upisujemo u svojstvo Text, a na nivou cijele alatne trake svojstvom TextAlign možemo podesiti kako će taj tekst biti prikazan – ispod sličice (Underneath) ili desno od nje (Right). Naime, za gumbe u alatnoj traci očekuje se da imaju sličice; čak i u slučajevima kad ih ne definirate, za njih će biti "rezerviran" prostor na gumbu.

Sličice (ikone) postavljamo na već viđeni način, pomoću komponente ImageList. U nju spremimo sve sličice koje će nam trebati, povežemo je s kontrolom ToolBar preko svojstva ImageList i onda u svakom gumbu definiramo indeks sličice koji će on koristiti, u svojstvu ImageIndex.

Tooltipove definiramo također za svaki gumb zasebno, upisujući ga u svojstvo ToolTipText, dok na razini cijele trake s alatima određujemo hoće li se oni i prikazivati, pomoću svojstva ShowToolTips.

Veličina alatne trake određuje se automatski, ako je svojstvo AutoSize uključeno. Ako nije, onda se veličina neće prilagođavati veličini gumba. Želite li pak utjecati na veličinu gumba, možete kroz svojstvo ButtonSize podesiti veću vrijednost od one automatski određene na temelju veličine ikona i teksta. Manju ne.

Dodjeljivanje funkcionalnosti gumbima

Tužna vijest koju saznate neposredno prije nego što želite dodijeliti kakvu funkcionalnost gumbima u alatnoj traci jest da gumbi nemaju vlastite događaje, već se sve obavlja preko događaja vezanih uz samu alatnu traku. Preciznije rečeno, radi se o događaju ButtonClick, koji nastupi kada kliknemo na jedan gumb, bez obzira koji. Drugim riječima, zadatak nam je otkriti koji je gumb pritisnut i sukladno tome izvršiti adekvatnu funkcionalnost. Evo kako to izgleda u praksi:

```
private void toolBar1_ButtonClick(object sender,
        System.Windows.Forms.ToolBarButtonClickEventArgs e)
```

```
{
    if (e.Button == toolNovi)
    {
        menuNovi.PerformClick();
    }
    else if (e.Button == toolOtvori)
    {
        menuOtvori.PerformClick();
    }
    else if (e.Button == toolSnimi)
    {
        menuSnimi.PerformClick();
    }
}
```

Primijetite da se funkcija kreirana na temelju događaja ButtonClick malo razlikuje od većine ostalih funkcija vezanih uz neki događaj. Naime, drugi parametar nije tipa System.EventArgs, već ToolBarButtonClickEventArgs, što u praksi znači da osim standardnih svojstava sadrži i svojstvo Button, koje nam kaže koji je gumb kliknut.

Stoga u tijelu funkcije koristimo naredbu *if* iza koje uspoređujemo je li gumb sadržan u svojstvu Button jednak nekom od gumba na našoj alatnoj traci (izrazi toolNovi, toolOtvori i toolSnimi su imena gumba na traci). Ukoliko jest, znači da je kliknut upravo taj gumb pa u bloku koji slijedi možemo odrediti funkcionalnost tog gumba. Mi u našem primjeru koristimo funkcionalnosti definirane za stavke izbornika pa metodom PerformClick() simuliramo klik na određenu stavku.

Statusna traka

Dok je alatna traka uvijek smještena na vrhu forme, na njezinom dnu ćemo u većini programa naći statusnu traku. Ona je najčešće informativnog karaktera, a što će se na njoj naći ovisi o namjeni programa odnosno njegovu autoru. Mi smo u našem primjeru odlučili na ovo mjesto zapisivati ime datoteke koju uređujemo, broj znakova koje datoteka sadrži (što je ujedno i veličina datoteke u bajtovima) te obavijest o tome je li datoteka snimljena ili nije.

Statusnu traku možemo koristiti na dva načina. Prvi način je znatno jednostavniji – cijela će se traka tretirati kao jedno polje i u nju ćete moći umetnuti samo jedan podatak. U tom ćete slučaju podatak ubaciti tako da ga upišete u svojstvo Text i on će biti prikazan na traci.

Složena statusna traka je po mnogočemu slična traci s alatima. Ona je podijeljena na više dijelova: kao što alatna traka ima gumbe, tako ova ima panele u koje možemo smještati informacije koje želimo prikazati. Takvih panela u jednoj traci može biti više, onoliko koliko nam treba, no da bi bili vidljivi treba uključiti svojstvo ShowPanels.

Slika 8-25:
Statusna traka s jednim podatkom i uključenim svojstvom SizingGrip

Neki se vizualni i funkcionalni parametri panela definiraju na nivou cijele statusne trake, dok se ostali podešavaju na nivou pojedinog panela. Naravno, statusna traka koristi velik broj već kod drugih kontrola spomenutih svojstava (Font, Cursor, Enabled, TabIndex, TabStop, Context Menu...) pa nam spomenuti ostaje samo svojstvo SizingGrip. Ono određuje hoće li u donjem desnom uglu trake biti prikazan trokutić koji sam po sebi nema funkciju, no trebao bi korisnika podsjetiti na mogućnost razvlačenja forme (vidi sliku 8-25).

Ne uključujte svojstvo SizingGrip na formama koje se ne mogu razvlačiti jer ćete zbuniti korisnika.

Slika 8-26:
Pomoćni prozor za dodavanje i uređivanje panela

Panele definiramo pomoću svojstva Panels iza kojeg se krije pomoćni prozor vrlo sličan onome koji smo sreli kod trake s alatima, pa ćemo spomenuti samo razlike.

Svaki panel na statusnoj traci može biti određene veličine. Veličinu u pikselima određujemo svojstvom Width, no samo ako je svojstvo AutoSize postavljeno na vrijednost None. Osim te vrijednosti, svojstvo AutoSize može biti Spring, što znači da će se panel toliko povećati koliko treba da bi zauzeo sav slobodni prostor statusne trake. Ukoliko više panela imaju AutoSize postavljen na Spring, onda će taj slobodni prostor međusobno ravnopravno podijeliti. Treća dostupna vrijednost svojstva AutoSize je Contents koja panelu nalaže da svoju širinu prilagođava sadržaju koji se u njemu nalazi. Međutim, kako širina pojedinog panela ne bi pala ispod određene vrijednosti, možemo podesiti svojstvo MinWidth kojim u pikselima određujemo najmanju dozvoljenu širinu panela.

Mi smo u našem primjeru prvi panel postavili na Spring, ostala dva na vrijednost Contents, a kozmetike radi podesili smo i minimalne širine. Također, dali smo im logična imena – statusImeDatoteke, statusSnimljeno i statusBrojZnakova.

Osim širine, panelu možemo određivati i izbočenost (svojstvo BorderStyle). Najčešći su uvučeni (Sunken) paneli, nešto češće se koriste ravni (None), a vrlo rijetko susrećemo izbočene panele (Raised). Primjere svih triju možete vidjeti na slici 8-27.

Slika 8-27:
Statusna traka s tri panela

U panele najčešće smještamo tekst (pogađate, svojstvo Text), a možemo postaviti i ikonu (svojstvo Icon). Hoće li tekst biti poravnan uz lijevi ili desni rub ili će biti centriran definiramo svojstvom Alignment.

Programiranje statusne trake

Iako statusna traka ima vlastite događaje, oni se rijetko koriste. Naime, kao što smo već spomenuli, ona se najčešće koristi za prikaz informacija, što znači da te informacije odnosno njihovu izmjenu moramo programirati na drugim mjestima u aplikaciji.

Kako bismo dovršili primjer Tekstualni editor, moramo popuniti statusnu traku trima informacijama. Prva je ime datoteke koju uređujemo, druga je status njezine snimljenosti i treća je broj znakova.

Želimo li u prvom panelu statusne trake imati uvijek aktualno ime datoteke, njegov sadržaj moramo osvježavati svaki put kada bi ime datoteke moglo biti promijenjeno. Promjena imena se može dogoditi prilikom stvaranja novog dokumenta, učitavanja i snimanja datoteke. Srećom, taj smo slučaj već imali prilikom spremanja imena datoteke u svojstvo textBox1.Tag tako da je dovoljno pronaći gdje sve pridružujemo vrijednom tom svojstvu i nakon njega dodati sljedeću liniju:

```
statusImeDatoteke.Text = textBox1.Tag;
```

Nemojte traženje dijelova kôda raditi ručno jer je jako vjerojatno da ćete nešto zaboraviti. Koristite vrlo bogate i moćne mogućnosti pretraživanja kôda i datoteka (u izborniku Edit, stavka Find and Replace).

Dosjetljiviji među vama uočit će da smo nakon ove intervencije dobili dva svojstva koja uvijek imaju istu vrijednost pa možemo zaključiti da jedno od njih zapravo i nije potrebno. Drugim riječima, možemo izbaciti korištenje svojstva textBox1.Tag za pamćenje imena datoteke i sve što smo radili s njim možemo raditi pomoću vrijednosti svojstva statusImeDatoteke.Text. (Isto će vrijediti i za varijablu Mijenjano.)

Sljedeći panel treba držati informaciju o snimljenosti datoteke. I taj smo slučaj već imali (sjetite se varijable Mijenjano), pa će i u ovom slučaju biti dovoljno pronaći sva pridruživanja toj varijabli i nakon toga dodati liniju kôda:

```
if (Mijenjano) statusSnimljeno.Text = "nije snimljeno";
          else statusSnimljeno.Text = "snimljeno";
```

Preostaje nam još zadnji panel, u kojem valja ispisati broj znakova. Ovo je najjednostavniji posao jer se broj znakova može promijeniti isključivo na jednom mjestu – prilikom promjene svojstva textBox1.Text. Kako postoji događaj vezan uz promjenu tog svojstva, naš jedini zadatak je u funkciju vezanu uz njega dodati podebljanu liniju kôda:

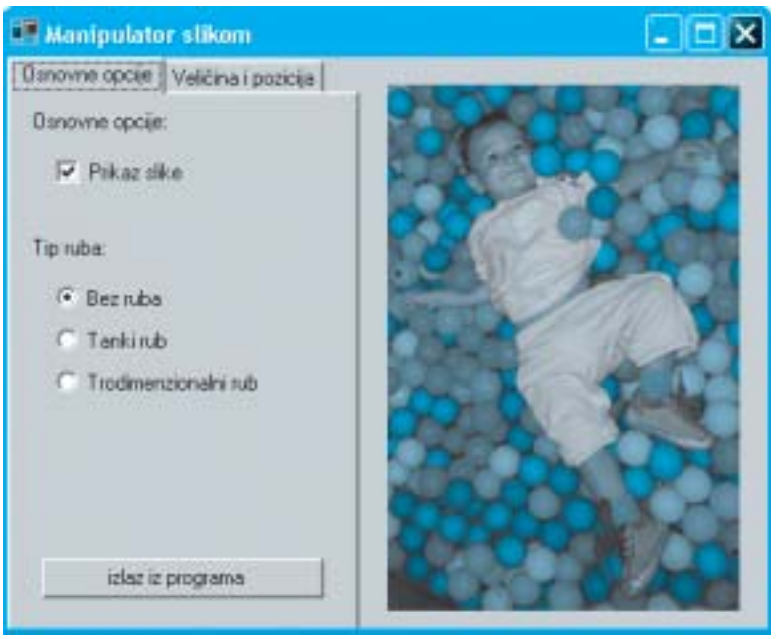
```
private void textBox1_TextChanged(...)
{
   Mijenjano = true;
   if (Mijenjano) statusSnimljeno.Text = "nije snimljeno";
              else statusSnimljeno.Text = "snimljeno";
   statusBrojZnakova.Text = textBox1.TextLength.ToString() + " zn";
}
```

Broj znakova, kao što vidite iz primjera, čitamo iz svojstva textBox1.TextLength koje zbog spajanja sa *stringom* " zn" i pridruživanja svojstvu statusBrojZnakova.Text (također tipa *string*) konvertiramo metodom ToString().

I to je to! Tekstualni editor je gotov.

Primjer: Manipulator slikom

Treći veliki primjer ovog poglavlja bit će aplikacija nazvana Manipulator slikom. Cilj aplikacije je pokazati još jednu skupinu kontrola koja se često koristi u prozorskim aplikacijama.

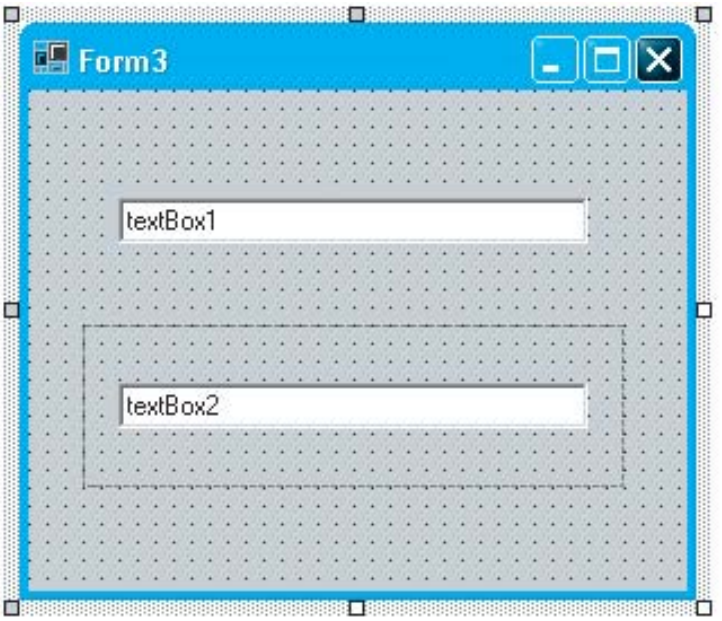


Slika 8-28: ***Osnovno sučelje aplikacije Manipulator slikom***

Za početak valja ukratko objasniti funkcionalnost aplikacije. Sučelje će biti podijeljeno u dva dijela – s desne ćemo strane prikazati sliku veličine 200x300 piksela, dok ćemo s lijeve strane kreirati niz opcija koje će utjecati na prezentaciju slike.

Paneli

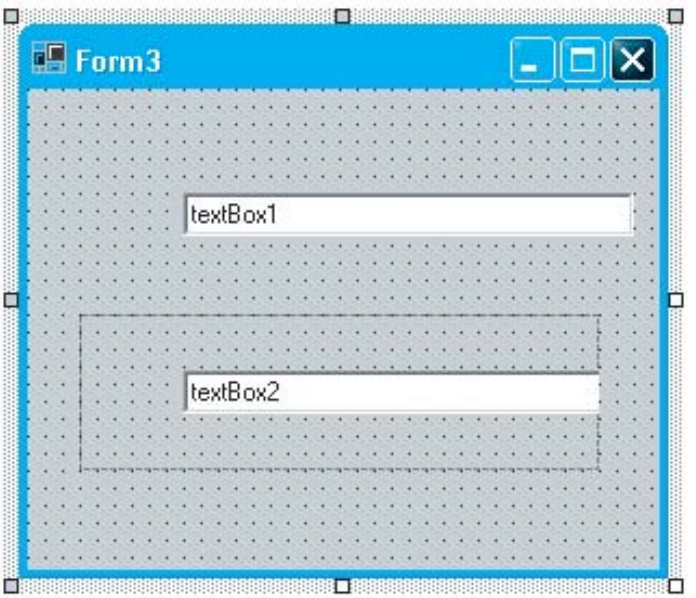
Prvo ćemo na formu dovući kontrolu Panel. To je najjednostavnija kontrola, koja praktički nema vlastitu funkcionalnost, nego služi isključivo ugošćavanju ostalih kontrola (služi kao spremnik, engl. *container)*. Najbolje je cijelu stvar objasniti primjerom. Pogledajte sliku 8-29.



Slika 8-29: ***Demonstracija uloge panela na formi***

Kada bismo pokrenuli ovu aplikaciju, sve što bi korisnik vidio jesu dva polja za upis jednake veličine, jednako udaljena od lijevog ruba prozora u kojem se nalaze. Međutim, u dizajnerskom načinu vidimo i pravokutnik iscrtkanog ruba koji nam ukazuje na postojanje panela odnosno činjenicu da se drugo polje za upis ne nalazi direktno na formi, već unutar tog panela.

Ako znamo da je između točkica na formi razmak od 8 piksela, jednostavno možemo izračunati poziciju kontrole textBox1 – ona se nalazi 56 piksela od lijevog ruba (svojstvo Left) i 40 piksela od gornjeg ruba forme (svojstvo Top). Na isti način bismo mogli izračunati i lokaciju kontrole textBox2, kada se ona ne bi nalazila u panelu. Međutim, kako se spomenuta kontrola nalazi u panelu, njezina pozicija se računa u odnosu na panel, a ne na formu. Zato je vrijednost njezine lokacije 8; 34.



Slika 8-30: ***Pomicanje kontrola unutar panela***

Pogledajmo sada sliku 8-30. S obje kontrole smo napravili istu stvar – pomaknuli smo ih za 40 piksela udesno, no samo je prva kontrola u cijelosti vidljiva, dok je dio kontrole koji izlazi iz granica panela nevidljiv. Proširimo li i panel udesno, i druga kontrola će se vidjeti u cijelosti.

Pomaknemo li cijeli panel, u odnosu na formu ćemo pomaknuti i sve kontrole koje se u njemu nalaze. Istina, njihove će brojčane vrijednosti lokacije ostati neizmijenjene, no kako su one relativne u odnosu na panel, promjena lokacije panela promijenit će tek njihovu apsolutnu poziciju (poziciju u odnosu na formu).

Iako vam se cijela priča oko panela vjerojatno čini banalnom i nepotrebnom, postoji velik broj situacija u kojima ove karakteristike značajno doprinose jednostavnosti i brzini razvoja aplikacija. U nastavku razrade primjera primijetit ćete nekoliko slučajeva, a u svojoj programerskoj karijeri naići ćete na barem još nekoliko.

Slike

Želite li na formi prikazati sliku, koristit ćete kontrolu PictureBox. Ona je znatno jednostavnija od kontrole ImageList i služi isključivo za prikaz slike na formi.

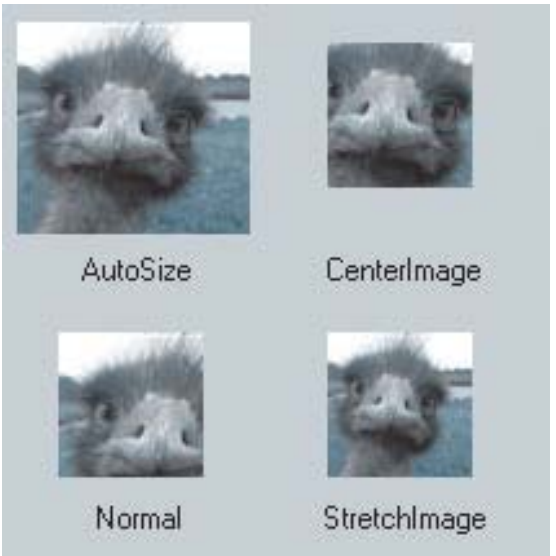
Jednu takvu kontrolu trebamo za naš primjer, no nećemo je direktno dovući na formu već ćemo je smjestiti unutar panela koji će zauzimati cijelu desnu polovicu forme (vidi sliku 8-28 – sivo područ-

je oko slike je panel). Zašto smo sliku smjestili u panel i sve tako posložili, bit će vam jasno nešto kasnije.

Dovlačenje kontrola na panel jednostavno je kao i dovlačenje na formu. Željenu kontrolu izaberete na Toolboxu i odvučete je na panel.

Za panel nije bitno kako se zove (ostavili smo automatski dano ime), jer mu nećemo u primjeru direktno pristupati, no sa slikom ćemo puno toga raditi, pa smo je odmah preimenovali u "slika" (bez navodnika).

Prazna kontrola tipa PictureBox nema smisla ako se u njoj ne nalazi slika, pa je preko svojstva Image valja ubaciti. Slika će gotovo sigurno biti drugačijih dimenzija od kontrole pa ih valja međusobno prilagoditi što je najlakše učiniti postavljanjem svojstva SizeMode na vrijednost AutoSize. To će automatski povećati (ili smanjiti) kontrolu kako bi mogla prikazati cijelu sliku. Svojstvo SizeMode može poprimiti i druge vrijednosti – Normal će ignorirati originalnu veličinu slike i prikazati samo onaj dio slike koji stane u dimenzije kontrole, CenterImage će imati isti efekt (osim što neće prikazati gornji lijevi kut slike već njezin centralni dio) dok će vrijednost StretchImage tako razvući sliku da se prilagodi dimenzijama kontrole (vidi sliku 8-31).



Slika 8-31:
Usporedba ponašanja svojstva SizeMode

Svi za jednoga, jedan za sve

Sustavno organizirani tipovi podataka u .NET-u često rezultiraju vrlo zanimljivim mogućnostima. Naime, često ćete shvatiti da se neki tip svojstva podudara s nekim drugim, što znači da ih međusobno možete uspoređivati i, što je često korisnije, pridruživati.

Kako otkriti kojeg je tipa neko svojstvo? Osim čitanja dokumentacije, tip svojstva možete otkriti tako da napišete ime tog svojstva u kodu, te zatim nad njim zadržite pokazivač miša nekoliko trenutaka. Primjerice, ako pokazivač miša postavite nad riječi Image u sljedećem izrazu...

```
pictureBox1.Image
```

...otkrit dobit ćete sljedeću informaciju:

```
System.Drawing.Image PictureBox.Image
```

Iz nje možete iščitati tip svojstva (prvi dio) i tip kontrole kojem pripada (drugi dio).

Ako znamo da su i slike u kolekciji imageList1.Images istoga tipa, onda možemo napisati sljedeće:

```
pictureBox1.Image =
imageList1.Images[0];
```

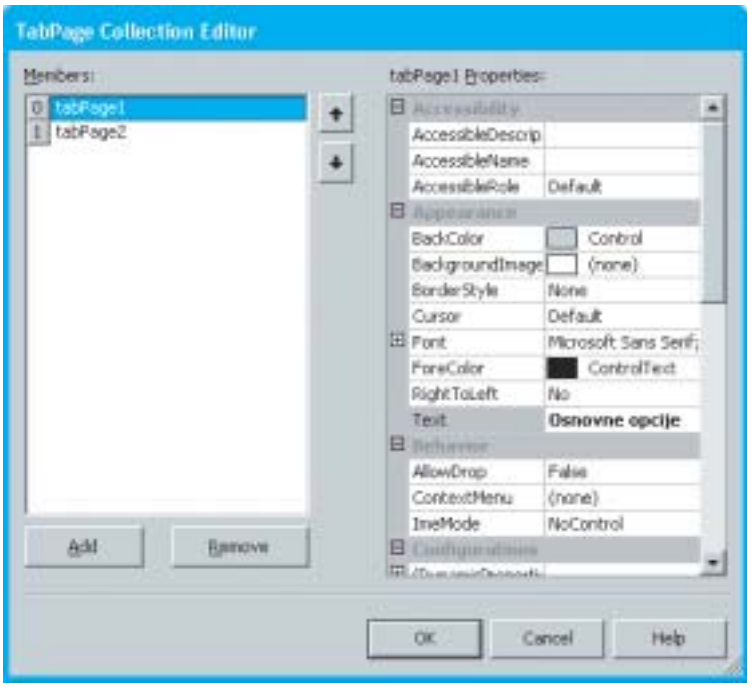
Taj izraz zamijenit će sliku u kontroli pictureBox1 prvom slikom u kolekciji imageList1.Images.

Kartice

Na lijevoj strani našeg primjera (slika 8-28) primjećujete dvije tzv. kartice nazvane "Osnovne opcije" i "Veličina i pozicija" koje su kreirane pomoću kontrole TabControl. Iz prakse korištenja računala sigurno znate čemu te kartice služe i kako se ponašaju. Iz programerskog aspekta radi se o nekoliko panela, među kojima se prebacujemo klikom na naslov one koju želimo prikazati.

Nakon dovlačenja kontrole TabControl na formu primijetit ćete da ne sadrži niti jednu karticu. Kartice dodajemo preko svojstva TabPages i pomoćnog prozora koji možete vidjeti na slici 8-32. Kako je svaka kartica zapravo panel (kontrola TabPage nasljeđuje velik broj svojih karakteristika od kontrole Panel), sve što smo u priči o panelima spomenuli vrijedi i ovdje.

Naravno, ima tu i nekih dodatnih mogućnosti. Primjerice, naslov kartice možemo odrediti kroz svojstvo Text, a moguće je i dodavati ikone (na već viđen način – roditeljsku kontrolu, koja je u ovom slučaju TabControl, povežemo s listom slika, a na nivou svake kartice upisujemo indeks ikone u toj listi).



Slika 8-32: ***Pomoćni prozor za upravljanje karticama u kontroli TabControl***

Vizualna svojstva (način prikaza, smještaj i slično) podešavamo većinom na roditeljskoj kontroli. Njima se ovaj puta nećemo baviti – ako vas zanima, poigrajte se samostalno.

Natpisi

Kontrola tipa Label vjerojatno je najjednostavnija kontrola koja postoji. Njezina glavna funkcija je ispisati neki tekst na ekranu iako bogatstvo svojstava i događaja koje sadrži omogućava da od nje napravite i mnogo više.

Mi ćemo spomenuti tek nekoliko osnovnih svojstava. U svojstvo Text unosite tekst koji želite da bude prikazan na ekranu. Svojstvom TextAlign određujete njegov smještaj u odnosu na površinu kontrole, a pomoću AutoSize uključujete automatsko prilagođavanje veličine kontrole sadržaju koji se u njoj nalazi. Naravno, tu je i cijela ergela već upoznatih svojstava: definiranje boja, dodavanje ikona, određivanje pokazivača miša, obrubi...

Na prvoj kartici kontrolu Label smo koristili dva puta kako bismo napisali natpise “Osnovne opcije” i “Tip ruba” (vidi sliku 8-28), a koristit ćemo je i na drugoj kartici (o tom potom).

Checkbox

Kontrolu CheckBox (u slobodnom prijevodu – kućicu s kvačicom) u primjeru koristimo za skrivanje i prikazivanje naše slike. Kao što smo već saznali, u tu se svrhu koristi svojstvo Visible pa

ćemo u funkciju vezanu uz događaj CheckedChanged kontrole koju smo nazvali checkPrikaz upisati sljedeći kôd:

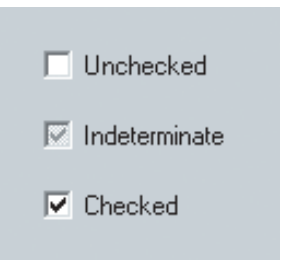
```
private void checkPrikaz_CheckedChanged(...)
{
    slika.Visible = checkPrikaz.Checked;
}
```

Uključenost odnosno isključenost kontrole CheckBox čita se iz svojstva Checked. Kad se vrijednost tog svojstva promijeni (dakle, kada korisnik klikne na kontrolu i promijeni joj stanje), poziva se gornja funkcija i prikaz slike (svojstvo Visible) se isključuje ukoliko je CheckBox isključen, i obrnuto.

U našem je primjeru (i velikoj većini ostalih aplikacija) kućica s kvačicom smještena lijevo od teksta kontrole (pogađate, svojstvo Text). To je moguće promijeniti, i to pomoću svojstva CheckAlign, koje omogućava čak devet pozicija kućice u odnosu na tekst.

Slično kao kod gumba, i ovdje je moguće koristiti modernije (Flat i Popup) vrijednosti svojstva FlatStyle, a cijelu je priču moguće začiniti i slikom (svojstvo Image), cijelom listom slika (ImageList i ImageIndex – praktično je kada želite promijeniti sliku ovisno o uključenosti kontrole) te smještajem slike u odnosu na kontrolu (ImageAlign).

Kontrola CheckBox inicijalno je postavljena da klikom na nju automatski mijenja svoje stanje. Međutim, takvo je ponašanje moguće i isključiti preko svojstva AutoCheck. U tom ćete slučaju sami, pomoću događaja Click, morati isprogramirati što će se dogoditi kada korisnik klikne na kontrolu.



Slika 8-33:
Tri stanja kontrole CheckBox

Osim uključenosti i isključenosti, kontrola CheckBox može biti konfigurirana da poprimi i tzv. neodređeno stanje (vidi sliku 8-33). Da bismo to postigli, trebamo uključiti svojstvo ThreeState. Nakon toga stanje kontrole nećemo provjeravati preko svojstva Checked, već CheckState koje može poprimiti vrijednosti Unchecked, Checked i Indeterminate (u kodu ih referenciramo kao CheckSta-

te.Indeterminate). Također, događaj na koji se vežemo očekujući promjenu je CheckStateChanged.

Radiobutton

Svojevrsna blizanka kontrole CheckBox je RadioButton. Osim vizualne razlike (pravokutna prema okrugla), puno je važnija funkcionalna. Naime, dok *CheckBoxovi* rade samostalno, *radiobuttoni* surađuju s ostalim kontrolama svog tipa i dopuštaju da istovremeno izaberete samo jednu među njima. Izuzmemo li još i mogućnost trostrukog stanja, kontrola RadioButton je u svemu ostalom jednaka CheckBoxu.

Lijepo ime *radiobutton*...

Proučavanje imena stvari i tehnologija, posebno onih nastalih prije komercijalizacije naše industrijske grane, može biti vrlo zanimljivo jer nerijetko otkrijete neočekivane tokove misli onih koji su ih imenovali. Jednim od takvih imena može se podičiti i kontrola RadioButton.

Sjećate li se starih radioprijemnika za automobile koji su imali nekoliko gumba za prebacivanje između predefiniranih radijskih postaja? Kad ste kliknuli na jedan od njih, on bi ostao pritisnut, stanica bi se prebacila, a ostali bi gumbi iskočili u svoje izbočeno stanje. Taj koncept i danas postoji na gotovo svim radioprijemnicima, jer je sasvim logično da ne možete odabrati više radiopostaja odjednom, samo što su izbočeno-uvučeni gumbi zamijenjeni LE-diodama i/ili prikazom broja stanice na zaslonu.

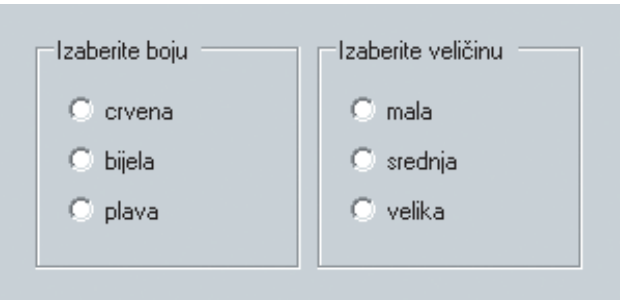
Zaključak okvira nazirete i sami – *radiobuttoni* dobili su ime po tim radijskim gumbima. Ipak, kako je takvo imenovanje prilično neintuitivno, u literaturi namijenjenoj korisnicima aplikacija sve se češće koristi pojam *option button*.

Pitanje koje se samo po sebi nameće glasi: što ukoliko na nekoj formi trebamo više skupina *radiobuttona*? Tu u igru dolaze paneli (i svi drugi slični oblici spremnika poput kontrola TabPage ili GroupBox) koji ograničavaju međusobnu suradnju *radiobuttona*. Drugim riječima, grupiranje *radiobuttona* i njihovo odvajanje od ostalih skupina možete riješiti tako da ih stavite u zaseban spremnik.

Obrada promjene odabranog radiobuttona

Kod promjene vrijednosti *radiobuttona*, za razliku od *checkboxa*, ne zanima nas status svakog pojedinog *radiobuttona* u skupini, već samo koji je od njih uključen. Kako je svaki *radiobutton*

zaseban objekt, ne postoji događaj za slučaj promjene vrijednosti bilo kojeg *radiobuttona*, no možemo ga simulirati.



Slika 8-34: ***Dvije skupine radiobuttona na istoj formi, svaka u svojoj kontroli tipa GroupBox***

Iskoristimo za to naš primjer. U prvu karticu (vidi sliku 8-28) dodajmo tri *radiobuttona* i imenujmo ih radioBezRuba, radioTankiRub i radio3DRub. Oni će služiti za mijenjanje obruba slike koja se nalazi s desne strane.

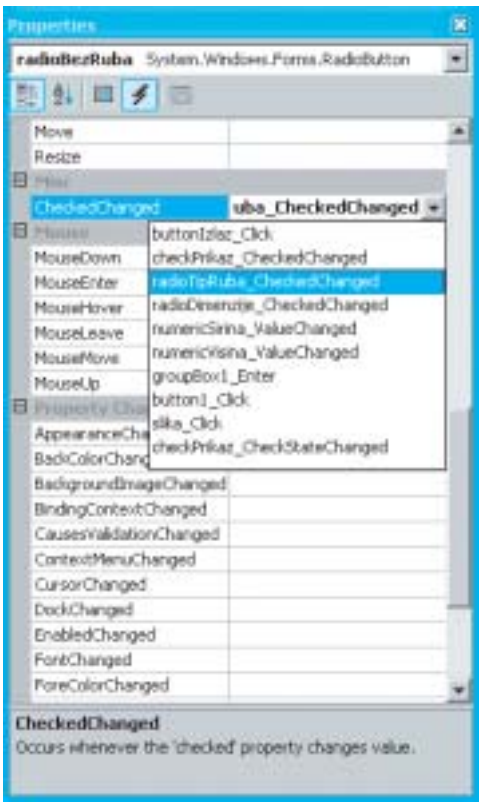
Označite prvu kontrolu i prebacite se u pomoćnom prozoru Properties na listu događaja te dvokliknite na događaj CheckedChanged. Otvorit će vam se funkcija koju preporučujemo da u kodu preimenujete jer trenutno u sebi sadrži ime kontrole kojoj pripada, a uskoro ćemo je povezati i s druge dvije. Mi smo je nazvali radioTipRuba_CheckedChanged. Nakon preimenovanja potrebno je vratiti se u dizajnerski način te i kod događaja CheckedChanged izmijeniti ime funkciji. To ne morate učiniti ručno, već pomoću padajućeg izbornika (vidi sliku 8-35).

Sada istu stvar treba napraviti i za ostale dvije kontrole, radioTankiRub i radio3DRub. Naravno, kod njih preskačete dvoklik za stvaranje nove funkcije; samo iz padajućeg izbornika izaberete istu funkciju koju ste izabrali i za kontrolu radioBezRuba. Rezultat – svaki put kada se promijeni svojstvo CheckedChanged neke od kontrola bit će pozvana ista funkcija, funkcija radioTipRuba_CheckedChanged.

U toj funkciji moramo provjeriti koji je *radiobutton* nakon promjene označen i sukladno tome promijeniti tip obruba slike (svojstvo slika.BorderStyle). Evo kako to izgleda u praksi:

Spomenuti "trik" s pridruživanjem iste funkcije većem broju kontrola ne pali uvijek. Postoje slučajevi kada je bitno da svaki radiobutton pokreće drugu funkciju, bez obzira na to što su povezani.

```
private void radioTipRuba_CheckedChanged(...)
{
    if (radioBezRuba.Checked)
    {
        slika.BorderStyle = BorderStyle.None;
    }
    else if (radioTankiRub.Checked)
    {
        slika.BorderStyle = BorderStyle.FixedSingle;
    }
    else if (radio3DRub.Checked)
    {
        slika.BorderStyle = BorderStyle.Fixed3D;
    }
}
```



Slika 8-35:
Padajući izbornik koji omogućava povezivanje događaja s bilo kojom postojećom funkcijom koja ima iste parametre

Gornji primjer nije dobar iako bez problema odrađuje ono što smo željeli. Zašto? Zato što se klikom na jedan *radiobutton* zapravo mijenjaju vrijednosti dvaju *radiobuttona* – onaj na koji kliknemo promijenit će svojstvo Checked na *true*, dok će onom koji je dotada bio izabran svojstvo Checked biti postavljeno na *false.* S obzirom na to da se promjena dogodila na dva mjesta, funkcija će biti pozvana dva puta. To u našem malom primjeru nema velikih negativnih posljedica jer je funkcionalnost vrlo jednostavna, no kod složenijih će primjera to biti uzaludno trošenje sistemskih resursa. Zato je sljedeći primjer puno primjereniji:

```
private void radioTipRuba_CheckedChanged (object sender, System.EventArgs e)
{
   RadioButton kliknut = (RadioButton) sender;
   if (kliknut.Checked)
   {
      if (kliknut == radioBezRuba)
      {
         slika.BorderStyle = BorderStyle.None;
      }
      else if (kliknut == radioTankiRub)
      {
         slika.BorderStyle = BorderStyle.FixedSingle;
      }
      else if (kliknut == radio3DRub)
      {
         slika.BorderStyle = BorderStyle.Fixed3D;
      }
   }
}
```

Primjećujete da po prvi put proučavamo funkciju prvog parametra funkcije vezane na događaj. Taj prvi parametar je tipa *object,* što znači da može sadržavati bilo kakvu vrijednost. U našem slučaju on sadrži onaj *radiobutton* čiji je događaj inicirao pozivanje funkcije. To vrijedi i općenito – parametar *sender* sadrži onu kontrolu kojoj pripada događaj koji je pozvao funkciju.

Međutim, kako je taj parametar tipa *object*, ne možemo ga direktno koristiti kao kontrolu (u našem slučaju *radiobutton*) već prije moramo izvršiti eksplicitnu konverziju (tzv. *castanje* o kojem smo govorili u petom poglavlju). To radimo u prvom redu funkcije – definiramo varijablu *kliknut,* tipa RadioButton, kojoj odmah pridružujemo vrijednost. Vrijednost pak dobivamo *castanjem* parametra *sender* u tip RadioButton. Nakon toga svako referenciranje varijable *kliknut* zapravo je referenciranje na *radiobutton* čiji je događaj pozvao funkciju.

Rekli smo da se funkcija poziva dvaput – jednom od strane označenog, drugi put od strane neoznačenog *radiobuttona.* Stoga, da bismo je izvršili samo jednom, dodali smo uvjet koji provjera-

va je li *radiobutton* koji je pozvao funkciju ovaj prvi (označen) te, ako jest, izvršavamo kôd unutar uvjeta. Ako nije, znači da je funkcija pozvana od neoznačenog *radiobuttona,* pa nema potrebe za (ponovnim) izvršavanjem koda.

Ako vas baš zanima koji će *radiobutton* prvi pozvati funkciju, točan odgovor je – onaj neoznačeni.

U drugom smo primjeru, u odnosu na prvi, izmijenili i ostale uvjete. Nije da "stari" nisu mogli ostati, već čisto da pokažemo kako je moguće uspoređivati dva objekta te i na taj način utvrditi koji je od potencijalnih kliknut).

Na kraju ove priče valja spomenuti da smo sličan efekt mogli postići i vezanjem na događaj Click. Štoviše, u tom slučaju funkcija ne bi bila pozivana dva puta jer će uvijek biti samo jedan *radio-button* na koji je kliknuto. Međutim, taj pristup ima dvije mane.

Prva je mogućnost da se vrijednost *radiobuttona* ne promijeni klikom već, primjerice, naredbom u kodu. U tom slučaju naša funkcija vezana uz Click ne bi bila pozvana i program ne bi radio kako očekujemo. Zato je izuzetno važno, ne samo u ovom primjeru nego i općenito, razmisliti na koji događaj vezati funkcionalnosti kako bi on obuhvatio sve promjene koje želimo obuhvatiti.

Druga mana je praktične prirode – da se nismo odlučili za ovakav pristup, ne bismo imali priliku pokazati *castanje* parametra *sender,* koje zna biti vrlo praktično u brojnim situacijama.

Brojčanik

Kako nam na ovoj kartici ne preostaje puno mjesta, prebacimo se na sljedeću (slika 8-36). Na njoj ćemo se igrati dimenzijama i pozicijom slikovne kontrole.

Prema slici 8-36 dovucite još četiri *radiobuttona* i dodijelite im, radi lakšeg snalaženja, imena radioAutoVelicina, radioRazvucena, radioNormalna i radioCentrirana. Osim toga, trebaju nam i dva brojčanika koja se skrivaju iza kontrole imena NumericUpDown. Njih nazovite numericSirina i numericVisina. (Mi smo dodali i pokoju kontrolu tipa Label, čisto radi jasnoće sučelja.)

Upravo će se dodatni *radiobuttoni* brinut za svojstvo SizeMode koje pripada kontroli slika. Funkcionalnost mijenjanja svojstva sada znate napraviti i sami (vidi priču o mijenjanju obruba) pa će funkcija (pridružena događajima CheckedChanged svih četiriju *radiobuttona)* izgledati ovako:

Slika 8-36:
Druga kartica primjera Manipulator slikom

```
private void radioDimenzije_CheckedChanged
                    (object sender, System.EventArgs e)
{
   RadioButton kliknut = (RadioButton) sender;
   if (kliknut.Checked)
   {

      // tu ćemo dodati još kôda (A)

      if (radioAutoVelicina.Checked)
      {
         slika.SizeMode = PictureBoxSizeMode.AutoSize;
         // tu ćemo dodati još kôda (B)
      }
      else if (radioRazvucena.Checked)
      {
         slika.SizeMode = PictureBoxSizeMode.StretchImage;
      }
      else if (radioNormalna.Checked)
      {
         slika.SizeMode = PictureBoxSizeMode.Normal;
```

```
            }
            else if (radioCentrirana.Checked)
            {
                slika.SizeMode = PictureBoxSizeMode.CenterImage;
            }
        }
    }
```

Međutim, kada je odabrana vrijednost AutoSize, onda nam brojčanici za određivanje dimenzija kontrole nisu potrebni, pa ih treba zasiviti. Drugim riječima, omogućenost brojčanika (numericSirina.Enabled) treba biti inverzno vrijednosti od označenosti *radiobuttona* radioAutoVelicina (radioAutoVelicina.Checked). Dodajmo stoga u kôd na mjesto označeno komentarom (A) sljedeće linije:

```
numericSirina.Enabled = !(radioAutoVelicina.Checked);
numericVisina.Enabled = !(radioAutoVelicina.Checked);
```

Na mjesto (B) treba ubaciti kôd koji će sinkronizirati vrijednosti u brojčanicima sa stvarnom veličinom kontrole sa slikom nakon odabira automatskog određivanja veličine kontrole. Naime, već smo spomenuli da se veličina kontrole, ako je svojstvo SizeMode postavljeno na vrijednost AutoSize, automatski prilagođava veličini slike, što znači da će dimenzije kontrole biti izmijenjene. Kako naši brojčanici prikazuju te dimenzije, nakon što se dogodi automatsko prilagođavanje treba popraviti vrijednosti u njima. To radimo na sljedeći način:

```
numericSirina.Value = Convert.ToDecimal(slika.Size.Width);
numericVisina.Value = Convert.ToDecimal(slika.Size.Height);
```

Tu dolazimo do prvog nepoznatog svojstva kontrole NumericUpDown, svojstva Value. Kao što vidite iz primjera, radi se o vrijednosti tipa *decimal,* što znači da prije pridruživanja vrijednosti slika.Size.Width i slika.Size.Height (koje su tipa *int*) moramo napraviti konverziju pomoću metode u klasi Convert (vidi peto poglavlje).

Kontrole NumericUpDown, kao što možete vidjeti, služe za upis ili odabir neke brojčane vrijednosti. Vrijednosti koje može poprimiti definiramo pomoću nekoliko svojstava. Broj decimalnih mjesta upisujemo u svojstvo DecimalPlaces. Minimalnu i maksimalnu vrijednost u svojstva Minimum i Maximum, dok korak povećanja (koji će nastupiti kada kliknemo na gumbiće za povećanje odnosno smanjenje vrijednosti u kućici) podešavamo pomoću svojstva Increment.

Najkorišteniji događaj ove kontrole je ValueChanged koji, pogađate, nastupa kada se promijeni vrijednost svojstva Value, bez obzira je li do nje došlo pritiskom na gumbiće, ručnim unosom broja ili promjenom iz kôda.

U primjeru uz svaki brojčanik vežemo zasebnu funkciju u kojima također moramo raditi konverziju, ovoga puta u drugom smjeru:

```
private void numericSirina_ValueChanged(...)
{
    slika.Width = Convert.ToInt16(numericSirina.Value);
}

private void numericVisina_ValueChanged(...)
{
  slika.Height = Convert.ToInt16(numericVisina.Value);
}
```

Ostale kontrole

Kontrola, dakako, ima još. Nažalost, zbog ograničenog prostora i brojnih tema kojih se još moramo dotaknuti, ne preostaje nam ništa drugo, nego ih samo ukratko spomenuti (tablica 8-2) i navesti čemu služe, a njihovo korištenje i proučavanje ostaviti vama. Podsjećamo da se reference svih kontrola nalaze u MSDN Library, na putanji .NET Development > .NET Framework SDK > NET Framework > Reference > Class Library > System.Windows.Forms.

Dodavanje novih kontrola

Iako kontrola koje dolaze unutar .NET Frameworka ima mnogo, kad-tad ćete poželjeti koristiti neku novu, koja se ne nalazi u originalnoj postavi. Osim što ih možete sami napraviti, možete ih i skinuti s Interneta i, ovisno o proizvođaču, besplatno ili uz naknadu koristiti u svojim aplikacijama.

Svaka kontrola dolazi s vlastitom dokumentacijom, prema kojoj možete shvatiti kako se koristi i kako radi. Međutim, dodavanje kontrola radi se uvijek na isti način.

Tu se vraćamo na priču o *assemblyjima.* (Sjeća li se tko drugog poglavlja?) Naime, svaka kontrola je zapravo *assembly* koji treba dodati na dohvat aplikacije u kojoj ga želimo koristiti. Za to imamo dva rješenja – smjestiti kontrolu u Global Assembly Cache ili u mapu *bin* aplikacije koju radimo.

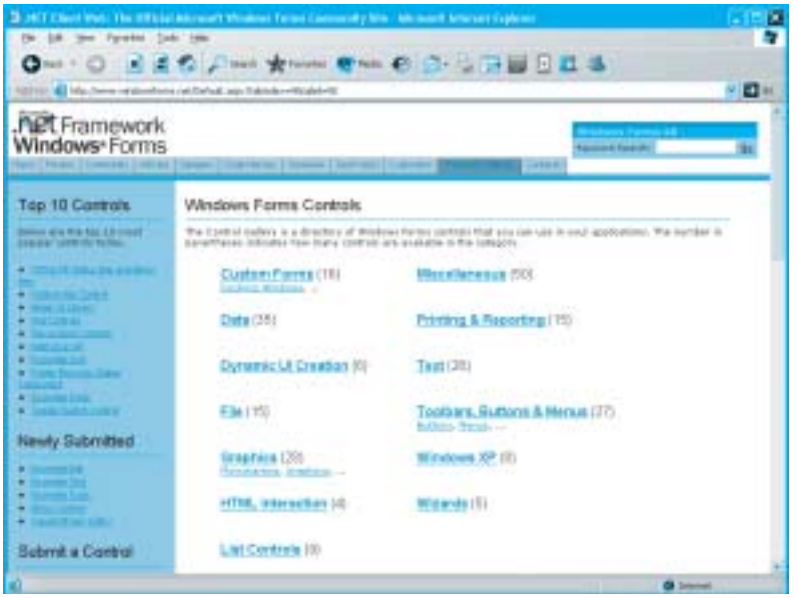
Naravno, zahvaljujući Visual Studiju, mi se tim "sitnicama" ne moramo zamarati. Kroz sljedeći ćemo primjer pokazati kako s Interneta skinuti, instalirati i koristiti neku kontrolu.

Za primjer smo odabrali skupinu kontrola nazvanih SandBar, koje možete preuzeti s adrese http://www.divil.co.uk/net/controls/sandbar/downloads.asp. Radi se o kontrolama za izradu atraktivnih traka s alatima i izbornika u stilu Microsoft Officea 2003.

U arhivu koju ćete skinuti sa spomenutih stranica bitna je samo jedna datoteka – ona s nastavkom ".dll". Ostale datoteke najčešće su dokumentacija i/ili konkretan primjer aplikacije u kojoj

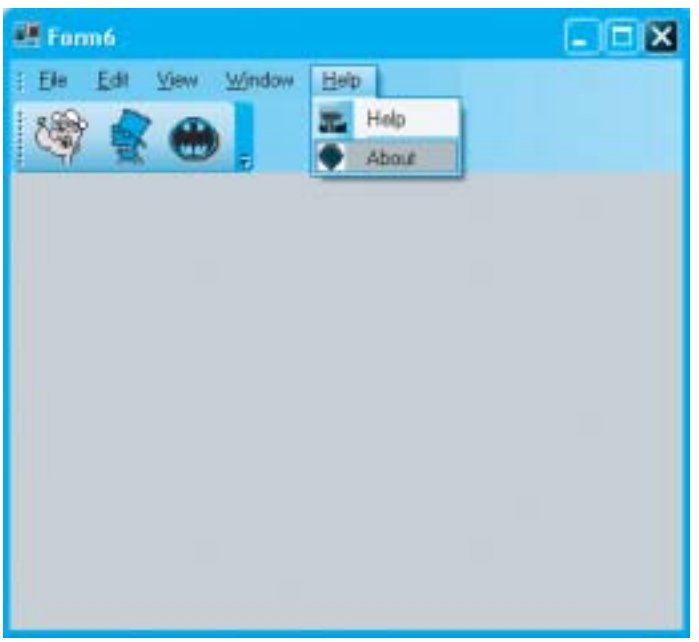
Tablica 8-2:
Popis nekih kontrola koje nismo uspjeli detaljnije obraditi u poglavlju. Neke od njih bit će objašnjene u narednim poglavljima.

Ime kontrole	Kratki opis
CheckedListBox	isto kao i ListBox, samo što svaki zapis ima
CheckBox	ColorDialog dijaloški okvir za izbor boje
DataGrid	tablica za prikaz podataka iz baze (više o tome u sljedećem poglavlju)
DateTimePicker	kućica za unos datuma s kalendarom
DomainUpDown	mješavina ComboBoxa i NumericUpDown – izbor *stringa* na način kontrole NumericUpDown
FolderBrowserDialog	dijaloški okvir za izbor mape na disku
FontDialog	dijaloški okvir za izbor fonta
HScrollBar	vodoravni kliznik
LinkLabel	label s mogućnošću pretvaranja u hiperlink
ListView	kontrola za prikaz stvari (poput mapa i datoteka u Windows Exploreru)
MonthCalendar	kalendar
NotifyIcon	ikona programa u *trayu* (kraj sata)
PageSetupDialog	dijaloški okvir za prilagođavanje stranice
PrintDialog	dijaloški okvir za ispis
PrintPreviewControl	kontrola za pregled dokumenta za ispis (PrintDocument)
PrintPreviewDialog	dijaloški okvir pregleda dokumenta prije ispisa
ProgressBar	prikaz napretka
RichTextBox	tekstualno polje za upis, s mogućnošću prikaza formatiranog teksta u formatu RTF
Splitter	razdjelnik u sučelju
Timer	štoperica; koristi se za ponavljanje određene radnje u određenom vremenskom periodu (svake sekunde, recimo)
ToolTip	pridruživanje *tipova* kontrolama koje ih inicijalno nemaju
TrackBar	traka s pomičnim klizačem
TreeView	stablasti prikaz strukture
VScrollBar	okomiti kliznik



Slika 8-37: ***Službena stranica zajednice Windows Forms na kojoj, između ostalog, možete pronaći i dodatne kontrole – adresa je http://www.windowsforms.net***

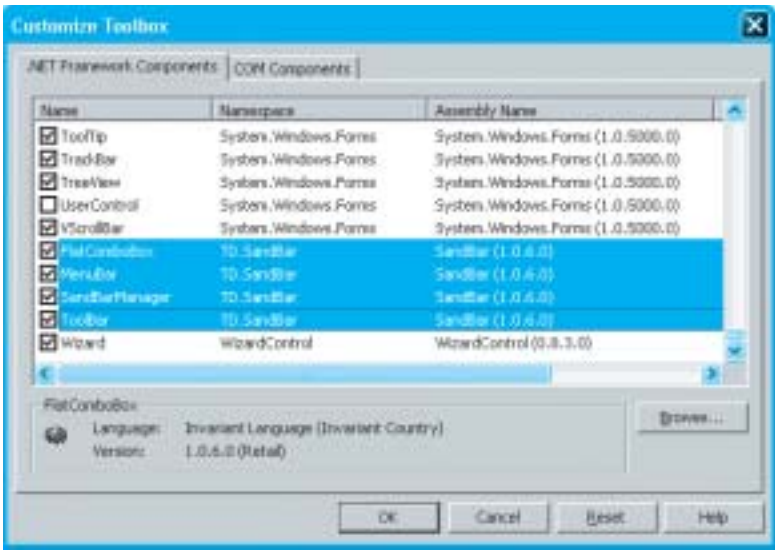
se kontrola koristi. (Ti primjeri ponekad su u Visual Basicu .NET pa vam u "prijevodu" puno može pomoći dvojezičnost trećeg poglavlja.)



Slika 8-38: ***Skupina kontrola SandBar omogućava puno efektnije izbornike i alatne trake od standardnih.***

Tu datoteku, koja se u našem slučaju zove SandBar.dll, treba smjestiti na neko sigurno mjesto gdje neće smetati. Predlažemo vam da otvorite posebnu mapu na disku u kojoj ćete držati komponente, najbolje zajedno s primjerima i dokumentacijom. Nebitno je gdje se ta mapa nalazi u odnosu na ostale datoteke, glavno da vi znate gdje je i kako do nje doći.

Sljedeći korak je dodavanje kontrola u pomoćni prozor ToolBox kako bi nam bile pri ruci kada ih trebamo. To radimo tako da kliknemo desnom tipkom miša negdje u tom prozoru i iz padajućeg izbornika izaberemo stavku Add/Remove Items. Otvorit će nam se prozor Customize Toolbox u kojemu valja kliknuti na gumb Browse i pronaći datoteku DLL koju smo maloprije smjestili na sigurno.



Slika 8-39: ***Prozor Customize Toolbox, koji služi za dodavanje kontrola u Toolbox***

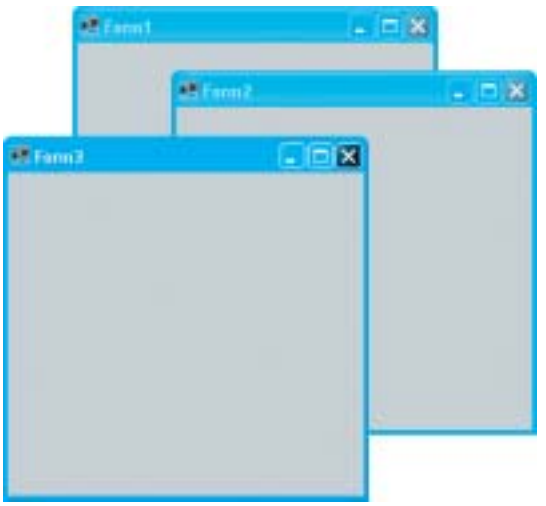
Nakon potvrde odabira, pojavit će nam se informacije o kontrolama koje su u toj datoteci pronađene. U našem se primjeru, kao što vidite na slici 8-39, radi o četiri kontrole koje su automatski označene i spremne za dodavanje na listu s ostalim kontrolama. Još pritisak na OK i možete ih početi koristiti...

Sve što dalje s njima radili izgledat će kao da se radi o najobičnijim kontrolama. Naravno, uvijek možete naići na kontrole koje nemaju dobru integraciju s Visual Studijem, no kod kvalitetnijih nećete imati takvih problema.

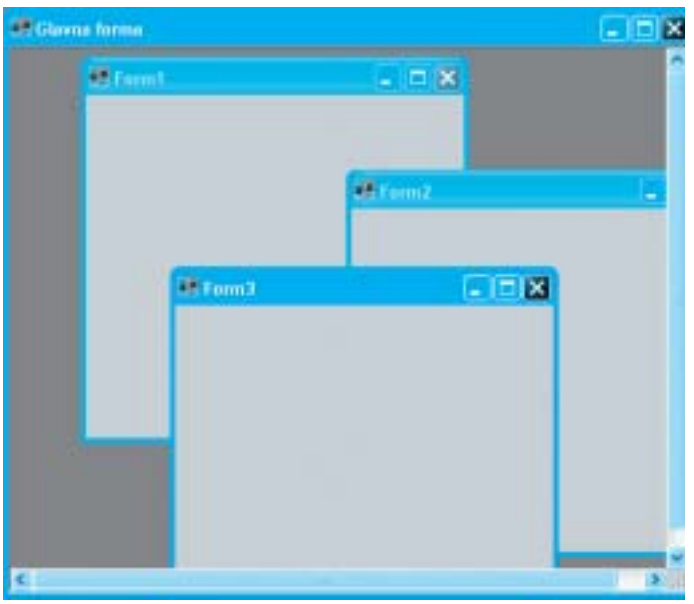
Svaki put kada odvučete kontrolu na formu, u pozadini će Visual Studio iskopirati datoteku s nastavkom ".dll" u mapu *bin* aplikacije u kojoj se koristi. Naravno, vaš jedini zadatak je da kod distribucije ne zaboravite spakirati i tu datoteku, a ne samo izvršnu datoteku aplikacije.

Rad s više formi

Rijetke su aplikacije koje imaju samo jednu formu. Makoliko primjer bio jednostavan, često će vam zatrebati još koja, bilo kao pomoćni prozor za neku radnju, bilo kao punopravni prozor aplikacije.



Slika 8-40: ***Single Document Interface***



Slika 8-41: ***Multiple Document Interface***

Ovisno o načinu otvaranja formi, aplikacija može imati Single Document Interface (SDI) ili Multiple Document Interface (MDI). U aplikaciji sa SDI-jem svaka je forma samostalna i prikazuje se u *taskbaru* operativnog sustava (iako se potonje, kao što smo već pokazali, dadne isključiti). Najbolji primjer ovakvog ponašanja je Internet Explorer – radi se o istoj aplikaciji, no kada otvaramo stranicu u novom prozoru, otvara se nova forma.

Primjer aplikacije sa MDI-jem je Microsoft Access. Učitamo li neku bazu i zatim otvorimo nekoliko njezinih tablica, svaka će se tablica otvarati u zasebnom prozoru (formi), no sve će se događati unutar klijentskog područja istoga, glavnog prozora. Taj glavni prozor naziva se spremnik MDI-ja (engl. *MDI container*).

Dodavanje i korištenje nove forme

Isprobajmo to na primjeru. Pretpostavimo da imate već napravljenu neku formu u jednoj aplikaciji i dođe vrijeme da dodate drugu. Jednostavno krenete u izbornik Visual Studija i pod glavnom stavkom Project odaberete Add Windows Form. Otvorit će vam se nova, prazna forma, kojoj možete dodavati kontrole i funkcionalnosti na jednak način kao što ste to radili s prvom.

Izvjesno je da ćete u nekom trenutku te forme željeti povezati odnosno da ćete iz prve željeti pozvati drugu. Recimo da ste za tu svrhu na prvoj postavili gumb klikom na koji bi druga forma trebala biti otvorena. Funkcija vezana uz događaj klika izgledat će ovako (naravno, isti kôd možete vezati uz bilo koji drugi događaj):

```
private void button1_Click(object sender, System.EventArgs e)
{
    Form2 Fo2 = new Form2();
    Fo2.Show();
}
```

Da biste razumjeli što smo upravo napravili, treba se sjetiti da slaganjem forme u Visual Studiju zapravo radimo novu klasu. Klasa (forma) koju smo preko izbornika dodali automatski je nazvana Form2. Sjetimo se također da je klasa samo nacrt za objekt koji će biti kreiran, pa stoga u prvoj liniji radimo upravo to – stvaramo objekt Fo2 na temelju nacrta klase Form2 (sintaksu za stvaranje objekta, vjerujemo, prepoznajete). Nakon što je objekt stvoren, treba ga prikazati, a to činimo metodom Show().

Dijaloški okviri su forme. Ugrađeni dijaloški okviri su forme s već gotovom funkcionalnošću, a forme koje otvaramo metodom ShowDialog() možemo zvati dijaloški okviri.

Otvaranjem nove forme na ovaj način primijetit ćete jednu zanimljivu karakteristiku. Naime, obje su forme ravnopravne i možete raditi malo u jednoj, malo u drugoj. Međutim, puno češće nam treba funkcionalnost kojom ćemo primorati korisnika da prvo obavi sav posao u novootvorenoj i da se tek onda smije vratiti na osnovnu. Takvo smo ponašanje već susreli kod dijaloških okvira (kod snimanja i učitavanja datoteka), a kod formi se postiže na isti način, istom metodom:

```
private void button1_Click(object sender, System.EventArgs e)
{
    Form2 Fo2 = new Form2();
    Fo2.ShowDialog();
}
```

Ako otvaramo formu metodom ShowDialog(), kôd nakon te metode neće biti izvršen sve dok korisnik ne zatvori otvorenu formu i vrati se na osnovnu. Ako koristimo metodu Show(), kôd će se izvršiti odmah po otvaranju forme, bez čekanja zatvaranja ili vraćanja na osnovnu.

Zatvaranje otvorene forme možete vršiti iz nje same, i to na sljedeći način:

```
Close();
```

Kao što smo već spomenuli, zatvaranje osnovne forme rezultirat će izlazom iz aplikacije.

Otvaranje formi u okruženju MDI nešto je složenije. Za početak, osnovnu formu moramo proglasiti spremnikom za MDI pomoću svojstva IsMDIContainer. Nakon toga treba kreirati novu formu (ili više njih, na isti način) te ih iz osnovne pozivati na sljedeći način:

```
private void menuNovi_Click(object sender, System.EventArgs e)
{
    Form2 Fo2 = new Form2();
    Fo2.MdiParent = this;
    Fo2.Show();
}
```

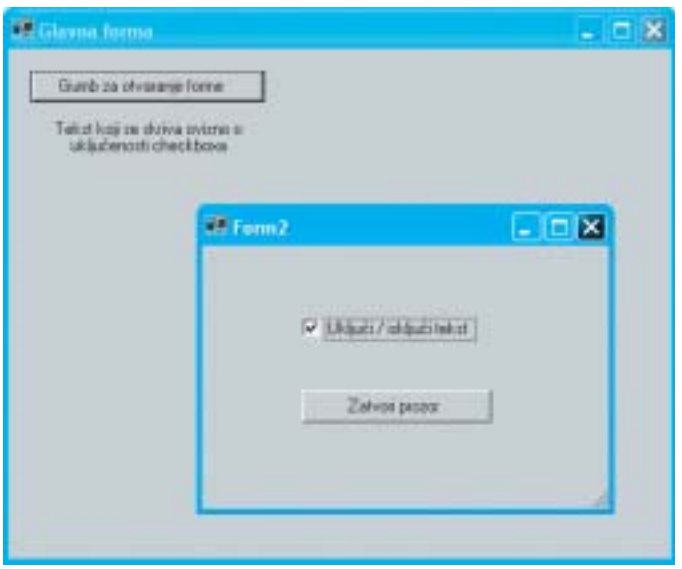
Primjećujete da smo dodali još jednu liniju kojom definiramo roditeljsku formu onoj koju otvaramo. Ključnom riječju *this* mislimo na formu u kojoj se trenutno nalazimo (dakle, kako formu otvaramo iz osnovne forme, mislimo na nju).

Dvojba koja se u ovom slučaju javlja jest na koji događaj povezati otvaranje forme u MDI okruženju. Naime, neozbiljno bi bilo staviti gumb (iako to, u testne svrhe, možete napraviti) jer će on biti prikazan iznad svih MDI-formi ili bilo kakvu drugu kontrolu tog tipa. Zato se najčešće u roditeljske MDI-forme stavljaju samo izbornik i statusna traka koje su pričvršćene uz rub forme (engl. *dock*) pa tako ne smetaju.

Prenošenje vrijednosti među formama

Kako sve forme pripadaju određenoj aplikaciji, vrlo je vjerojatno da će međusobno morati izmjenjivati određene podatke. To možemo postići na nekoliko načina, a mi ćemo vam pokazati najjednostavniji među njima.

Cijela priča temelji se na teoretskim osnovama obrađenima u petom i šestom poglavlju. Mi smo među iskoristivim rješenjima izabrali ono koje se temelji na javnim (*public*) članovima klase, konkretno varijablama.



Slika 8-42:
Primjer za demonstraciju prenošenja vrijednosti među formama

Uzmimo za primjer da imamo dvije forme – Form1 i Form2 te da je prva među njima osnovna. Na prvoj se nalazi gumb koji će nam poslužiti za otvaranje druge forme i kontrola tipa Label, a na drugoj se nalazi *checkbox* koji će služiti za pokazivanje i sakrivanje kontrole Label na prvoj formi. Da bismo mogli iz prve forme pristupiti stanju *checkboxa*, uvest ćemo javnu varijablu checkBoxVrijednost.

U kôd druge forme (Form2) dodat ćemo sljedeće (najbolje odmah nakon konstruktora klase):

```
public bool checkBoxVrijednost
{
    get
    {
        return checkBox1.Checked;
    }
    set
    {
        checkBox1.Checked = value;
    }
}
```

Riječju *public* određujemo da varijabla tipa *boolean* bude dostupna svima koji koriste tu klasu (vidi šesto poglavlje). Zatim ključnim riječima *get* i *set* definiramo kako će se varijabla ponašati kada čitamo njenu vrijednost (*get*) odnosno mijenjamo (*set*). Tako kažemo: kada netko bude tražio vrijednost naše varijable, vrati mu (*return*) vrijednost svojstva checkBox1.Checked. S druge strane, kada neko pridružuje novu vrijednost našoj varijabli, tu vrijednost (*value*) pridruži svojstvu checkBox1.Checked. Na taj način ćemo postići potpunu sinkroniziranost (javne) varijable checkBoxVrijednost i (privatnog) svojstva checkBox1.Checked.

Sve kontrole na formi automatski su dosega *private* i zato im nije moguće pristupiti iz ostalih klasa (formi).

Mogli smo izabrati i drugačiji pristup – varijablu checkBoxVrijednost samo definirati, a brigu o njezinoj vrijednosti prepustiti događajima kontrole checkBox1. Dakako, takav pristup bio bi znatno nespretniji.

Vratimo se sada na prvu, osnovnu formu i razmotrimo funkciju vezanu uz događaj klika na gumb:

```
private void button1_Click(object sender, System.EventArgs e)
{
    Form2 Fo2 = new Form2();
    Fo2.checkBoxVrijednost = label1.Visible;
    Fo2.ShowDialog();
    label1.Visible = Fo2.checkBoxVrijednost;
}
```

Prije nego što prikazujemo formu, namještamo vrijednost javne varijable pridružujući joj vrijednost svojstva label1.Visible. Ukoliko je kontrola label1 vidljiva, *checkbox* na drugoj kontroli će biti označen. Vrijednost će prvo biti pridružena varijabli checkBoxVrijednost, a zatim će, zbog takve definicije varijable, biti pridružena svojstvu checkBox1.Checked koji će utjecati na označenost te kontrole.

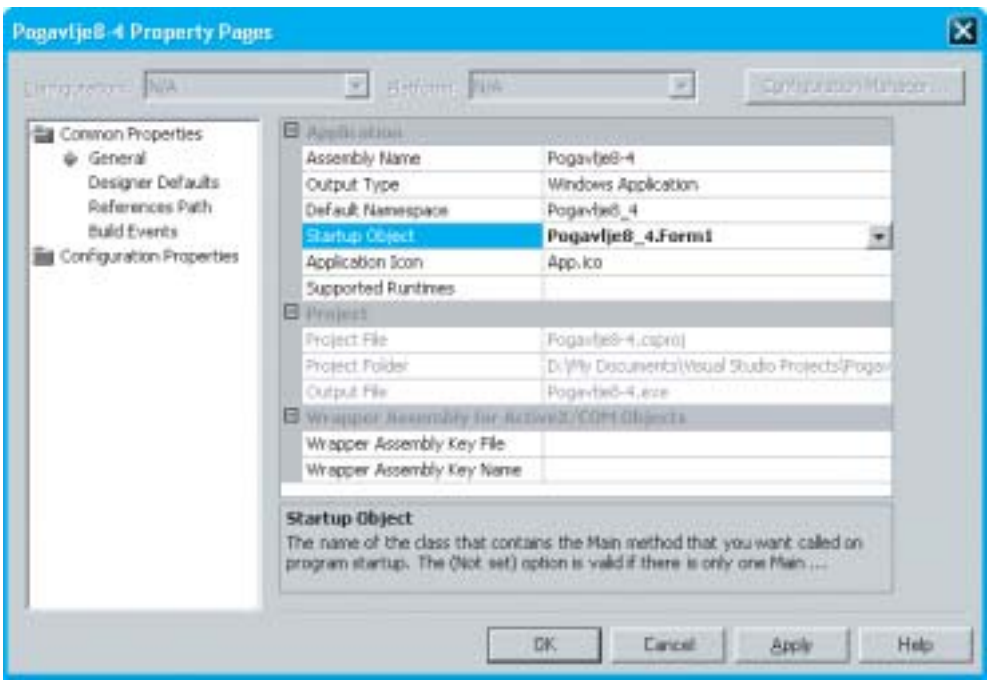
Nakon toga pozivamo formu u obliku dijaloškog okvira. Nakon što se korisnik poigra s *checkboxom* i zatvori formu, vraćamo se u gornji kôd i vrijednost varijable checkBoxVrijednost pridružujemo svojstvu label1.Visible. I ovdje se događa slična priča – u definiciji varijable odredili smo da vrijednost koja će biti vraćena zapravo bude uzeta iz svojstva checkBox1.Checked.

Načina za prenošenje vrijednosti među formama ima još. Možemo, primjerice, varijable prenositi kao parametre konstruktora klase, no to vam ostavljamo na samostalno proučavanje.

Promjena osnovne forme

Prilikom pokretanja aplikacije pokreće se metoda Main(). Ta se metoda automatski kreira prilikom otvaranja prve forme u projektu, koja ujedno postaje i osnovna forma. Metoda Main() izgleda ovako:

Slika 8-43:
Prozor za podešavanje svojstva Startup Object za slučaj postojanja više metoda Main() u aplikaciji



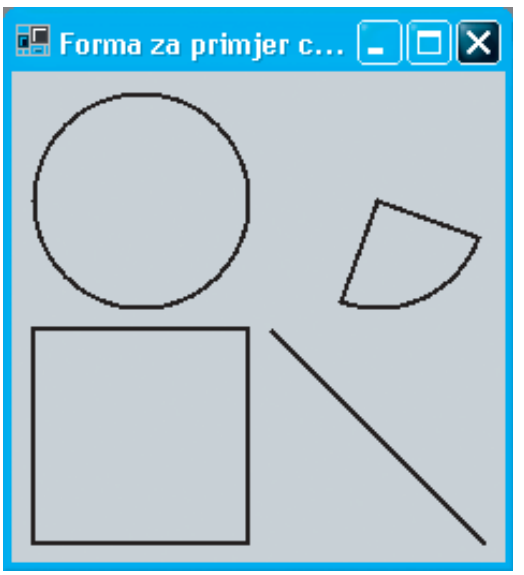
```
[STAThread]
static void Main()
{
    Application.Run(new Form1());
}
```

Želite li promijeniti formu koja se prva otvara, cijelu metodu Main (uključujući i izraz u uglatim zagradama) izrežite iz forme koja je dotad bila osnovna i prebacite je u drugu. Također, nemojte zaboraviti promijeniti ime forme koja se navodi kao parametar metodi Application.Run jer u protivnom nećete ništa postići.

Ako u cijeloj aplikaciji postoji samo jedna forma s metodom Main(), onda je sasvim jasno da će ona biti izvršena. Ukoliko pak više formi ima spomenutu metodu, onda ćete u izborniku, stavka Project > ImeProjekta Properties trebati odrediti vrijednost svojstva Startup Object. U protivnom će vam kompajler prijaviti grešku jer neće znati koja je od postojećih metoda prava.

Grafika

Postoji vjerojatnost da ni postojeće ni dodatne kontrole neće zadovoljiti vaše potrebe, posebno što se vizualnih zahtjeva tiče. Neke takve probleme moći ćete riješiti slikama, no ponekad će biti puno jednostavnije neke stvari na ekranu – nacrtati. O crtanju ovog tipa moglo bi se napisati cijelo poglavlje, no mi ćemo se ovdje osvrnuti tek na osnove osnova.



Slika 8-44:
Primjer crtanja na formi

Za crtanje po ekranu (i, kao što ćemo kasnije vidjeti, nekim drugim izlaznim jedinicama) brine se Graphical Device Interface (skraćeno GDI) koji u .NET-u dolazi u upravljanom i prilagođenom obliku nazvanom GDI+. To nam je sučelje dostupno preko *namespacea* System.Drawing.

Važno je znati da crtanje nije moguće u bilo koje vrijeme. Ne možete se u nekoj funkciji samo sjetiti nešto nacrtati, pozvati određenu metodu i očekivati da bude nacrtano. Sva se crtanja moraju vršiti prilikom iscrtavanja kontrole, što znači za vrijeme događaja Paint.

Evo primjera pomoću kojeg ćemo nacrtati niz geometrijskih oblika na formi, poput onoga na slici 8-44:

```
private void Forma_Paint (object sender, System.Windows.Forms.PaintEventArgs e)
{
  Pen crta = new Pen(Color.Black, 2);
  e.Graphics.DrawEllipse(crta, 10, 10, 100, 100);
  e.Graphics.DrawRectangle(crta, 10, 120, 100, 100);
  e.Graphics.DrawPie(crta, 120, 10, 100, 100, 20, 90);
  e.Graphics.DrawLine(crta, 120, 120, 220, 220);
}
```

Funkcija vezana uz događaj Paint kao drugi parametar ima objekt *e* tipa System.Windows.Forms.-PaintEventArgs. U njemu se, među ostalima, nalazi objekt Graphics preko kojega pristupamo površini i u kojem se nalaze metode za crtanje. U primjeru koristimo četiri najjednostavnije, a svaka od njih, osim potrebnih koordinata, kao prvi parametar ima varijablu tipa Pen koja određuje kako će izgledati crta kojom će oblik biti nacrtan. Mi smo se odlučili za crnu boju i širinu od dva piksela, no Pen skriva i druge, naprednije mogućnosti.

Također, svaka od metoda je preopterećena, što znači da ćete umjesto parametara iz primjera moći koristiti i neki drugi skup parametara. Mi u te sitnice ovdje nećemo ulaziti – dovoljno je u kodu započeti pisati "e.Graphics." i zahvaljujući IntelliSensu pojavit će se popis svih dostupnih metoda, a kasnije i popis parametara koje možete koristiti, sve s objašnjenjima.

Pozabavit ćemo se još dvjema metodama – jednom koja omogućava ispis slova u određenom fontu i drugom koja omogućava prikaz slike.

```
Font slova = new Font("Arial", 10);
Brush kist = Brushes.Green;
e.Graphics.DrawString("Microsoft .NET", slova, kist, 10, 230);
```

Jedna od kombinacija parametara metode DrawString koja služi za ispis teksta je ova iz primjera. Prvo navodimo *string* koji želimo ispisati, zatim tip slova, tip "kista" te na kraju koordinate na kojima želimo ispisati tekst.

Istu stvar možemo pisati i na sljedeći način, iako je on puno nepregledniji:

```
e.Graphics.DrawString("Microsoft .NET", new Font("Arial", 10),
      new SolidBrush(Color.Brown), 10, 230);
```

Metoda za prikaz slike izgleda ovako:

```
Image slika = Image.FromFile(@"D:\mapa\slika.bmp");
e.Graphics.DrawImage(slika, 230, 10);
```

Znak "\" je tzv. *escape*-znak, koji u kombinaciji s ostalim znakovima predstavlja neke specijalne znakove (sjetite se "\n" za prelazak novi redak). Kako se on koristi u definiranju putanja do određene mape, znakom "@" smo kompajleru dali do znanja da ga u *stringu* tretira normalno, a ne kao *escape*-znak.

Naravno, i ova metoda je preopterećena (i to čak 30 puta!) pa postoji velik broj načina na koji možete prikazati sliku, što vam i preporučujemo jer često ne možete biti sigurni u putanju datoteke.

Crtanje po kontrolama često se koristi kada se želi modificirati postojeća kontrolu odnosno na temelju postojeće napraviti vlastita. U tim slučajevima treba naslijediti neku kontrolu (najčešće će to biti osnovna, kontrola Control), prekoračiti metodu OnPaint i ubaciti svoje crtarije.

Ispis na pisač

Kod ispisa na pisač situacija nije tako bajna kao s izradom korisničkog sučelja. Kao što možete zaključiti po smještaju ovog odlomka, stvar se svodi na rad s GDI+-om. No, krenimo redom...

Za ispis na pisač potrebna nam je kontrola PrintDocument. Nakon što je dovučemo na kontrolu, bit će nam dostupan objekt printDocument1 koji ima događaje BeginPrint, EndPrint i PrintPage. Uz prva dva bismo definirali stvari koje treba podesiti na početku i na kraju printanja, a uz PrintPage treba nacrtati ono što želimo ispisati. Kao što ćete iz primjera što slijedi primijetiti, stvar je vrlo slična crtanju po kontrolama:

```
private void printDocument1_PrintPage
      (object sender, System.Drawing.Printing.PrintPageEventArgs e)
{
   Pen crta;
   crta = new Pen(Color.Black, 2);
   e.Graphics.DrawEllipse(crta, 10, 10, 100, 100);
   e.Graphics.DrawRectangle(crta, 10, 120, 100, 100);
   e.Graphics.DrawPie(crta, 120, 10, 100, 100, 20, 90);
   e.Graphics.DrawLine(crta, 120, 120, 220, 220);
   e.HasMorePages = false;
}
```

Jedina razlika među primjerima je posljednja linija, u kojoj koristimo svojstvo HasMorePages. Pomoću njega dajemo do znanja ima li još stranica nakon ove. Ako ima, događaj PrintPage nastupit će još jednom. Na vama je zadatak da se pobrinete da svaka sljedeća stranica bude drugačije ispisana. Jedno banalno i ne uvijek prihvatljivo rješenje tog problema može biti ovo:

```
private int stranica = 1;

private void printDocument1_PrintPage(...)
{
   if (stranica == 1)
   {
      // crtanje prve stranice
      e.HasMorePages = true;
   }
   else
   {
      // crtanje druge stranice
      e.HasMorePages = false;
   }
   stranica++;
}
```

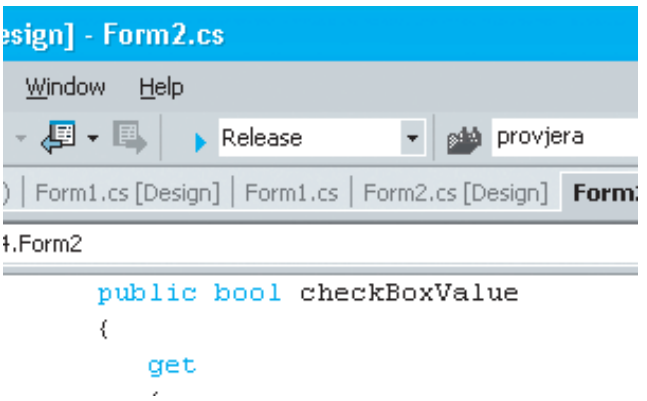
Kada prijeći na novu stranicu morat ćete izračunati sami. Naime, kako veličina papira u printeru može biti različita, tako se mijenja i površina na koju možete pisati. Veličinu cijelog papira možete dobiti kroz svojstvo e.PageBounds, dok su dimenzije te površine, umanjene za margine na koje ne možete pisati, zapisane u e.MarginBounds. Oba svojstva su tipa Rectangle, što znači da, između ostalog, možete koristiti sljedeće vrijednosti:

```
e.MarginBounds.Width;
e.MarginBounds.Height;
```

Naravno, crtanje i ispis skrivaju još velik broj tajni i trikova, za koje nažalost nemamo mjesta. Ipak, nadamo se da su vam ovi početnički primjeri pojasnili način na koji stvari rade i poslužili kao podloga za proučavanje dokumentacije ili primjera na Internetu.

Distribucija

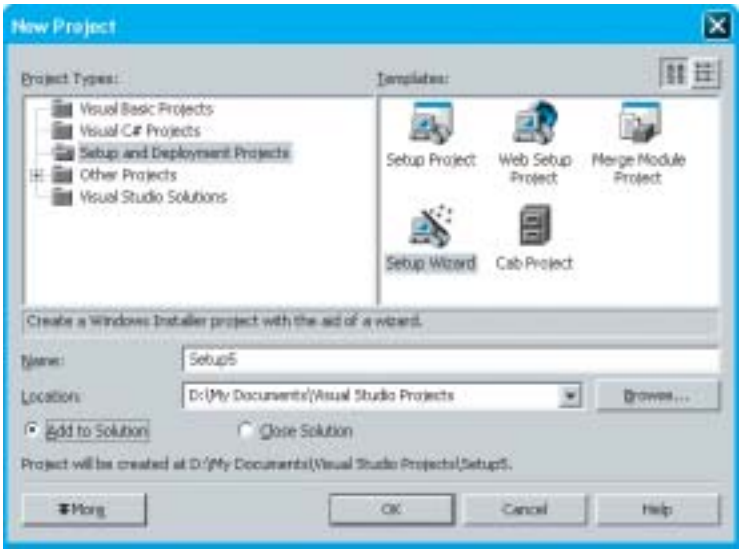
Nakon što napravite prozorsku aplikaciju, preostaje vam distribuirati je korisnicima. Najbolje kod aplikacija pisanih na platformi .NET je što u većini slučajeva možete jednostavno kopirati izvršnu datoteku s nastavkom ".exe" i eventualne dodatne datoteke koje koristite. Ipak, prije toga projekt morate kompajlirati u načinu Release (za razliku od načina Debug, koji je po *defaultu* aktivan). Takav način kompajliranja rezultirat će bržom i manjom aplikacijom jer neće sadržavati stvari potrebne za *debugiranje*. Prebacivanje možete napraviti u alatnoj traci Visual Studija, u padajućem izborniku, desno od ikone Start (vidi sliku 8-45).



Slika 8-45: ***Prebacivanje kompajliranja u način Release***

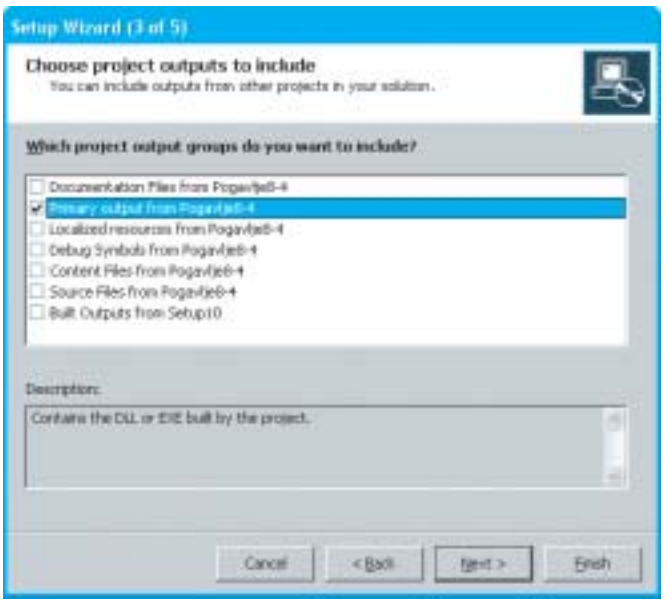
Međutim, ljudi su navikli na instalacijske procedure koje će im, osim kopiranja potrebnih datoteka, kreirati mape, prečace u popisu programa i slično. Ponekad prilikom instalacije treba napraviti i neke dodatne poslove, poput inicijalizacije baze podataka ili zapisivanja podataka u *registry*. Bilo kako bilo, u tome vam može pomoći kreiranje posebnog instalacijskog projekta.

Želite li kreirati instalacijski projekt za svoju aplikaciju, prvo je morate otvoriti i kompajlirati u načinu Release. Zatim treba kreirati novi instalacijski projekt, no tako da ga dodate u rješenje u kojem se nalazi aplikacija (vidi sliku 8-46). Tu postoji mogućnost odabira više tipova instalacija, a mi smo odabrali najjednostavniju koja uključuje instalacijskog čarobnjaka.



Slika 8-46: ***Kreiranje instalacijskog projekta (uočite odabranu opciju Add to Solution)***

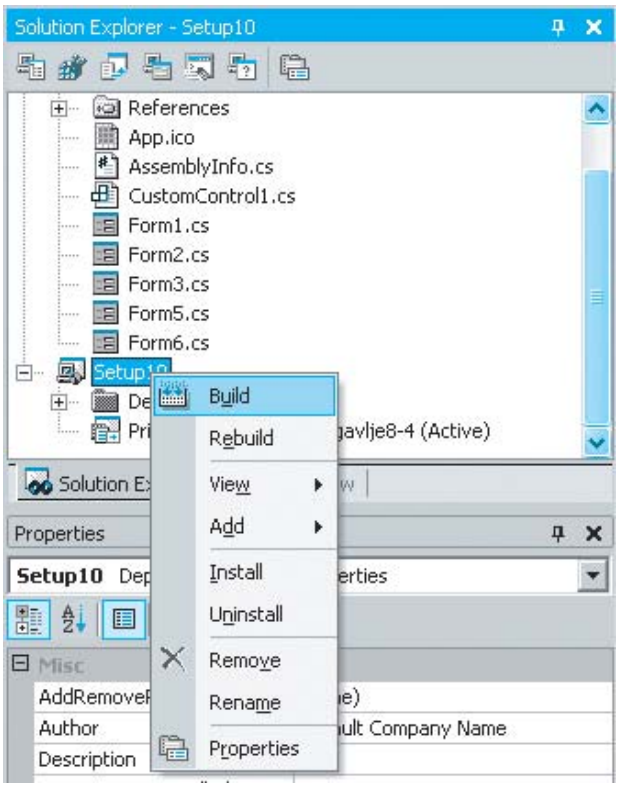
Nakon odabira instalacijskog projekta, pokrenut će se čarobnjak u kojem ćemo definirati određene parametre instalacije. Prvo biramo tip aplikacije, zatim projektne grupe koje ćemo unijeti u instalaciju (svakako treba izabrati Primary output koji uključuje izvršnu datoteku, a prema potrebi i ostale stavke), a na kraju nam se nudi mogućnost dodavanja ostalih datoteka (poput *readme*-dokumenata).



Slika 8-47: ***U instalaciju možete uključiti sve pa i izvorni kôd, no najčešće ćete uključiti samo Primary output.***

Slijedi igranje u sučelju pomoću kojeg možete dodati stavke u izbornik Programs, postavljati prečace na Desktop ili smještati datoteke u druge sistemske i programske mape.

U prozoru Solution Explorer treba označiti ime instalacijskog projekta kako bismo došli do njegovih svojstava. Među njima možemo pronaći mogućnost upisa imena aplikacije, inačice, autora, proizvođača i brojnih drugih parametara koji utječu na funkcionalan i vizualan način. Svakako im posvetite dovoljno vremena i upišite adekvatne vrijednosti.



Slika 8-48:
Solution Explorer i izbornik preko kojega kompajliramo i testiramo instalacijski projekt

Nakon što ste podesili sve parametre, vrijeme je da kompajlirate instalacijski projekt i isprobate ga. To ćete učiniti tako da u Solution Exploreru kliknete na instalacijski projekt desnom tipkom miša i odaberete stavku Build. Zatim u istom izborniku pronađite Install i vidite kako stvari funkcioniraju.

Kad budete zadovoljni instalacijskom procedurom, kompajlirajte projekt u načinu Release i potražite instalacijske datoteke koje je stvorio, a koje ćete vi distribuirati. One se nalaze u podmapi "Release", koja se pak nalazi u mapi instalacijskog projekta (ovisno o tome gdje ste ga smjestili prilikom kreiranja).

Slika 8-49: ***Radite li sustav pomoći za prozorske aplikacije, adresa koja vas zanima je http://www.mshelpwiki.com***

Osim izrade instalacijske procedure, svaka ozbiljna aplikacija zahtjeva sustav pomoći. Mi u tu problematiku nećemo ulaziti jer izlazi iz okvira programiranja, a one koje zanima upućujemo na internetsku adresu uz sliku.

Ima li još?

Naravno da ima. O programiranju prozorskih aplikacija može se još štošta reći, no zbog namjere da u knjizi pokrijemo sve aspekte programiranja u .NET-u ostaje dosta stvari koje ćete morati otkrivati sami. Srećom, sve se svodi na iste principe, tako da će vam primjeri koje smo u ovom poglavlju obradili, uz dozu iskustva i redovito druženje s dokumentacijom, omogućiti da riješite i one probleme kojih se ovdje nismo dotaknuli. Sretno!

9. POGLAVLJE

ADO.NET

U ovom poglavlju:

- Osnove jezika SQL
- naredbe SELECT, INSERT, UPDATE i DELETE
- Arhitektura ADO.NET-a
- Osnovne komponente ADO.NET-a
- Razlika izmedu *DataReadera* i *DataSeta*
- Praktičan rad s ADO.NET-om
- Prikaz podataka u kontroli *DataGrid*
- Korištenje *DataReadera* za dohvat podataka
- Stvaranje i izvršavanje objekata *Command*

U prethodnom poglavlju naučili ste izrađivati jednostavne aplikacije, a sad idemo korak dalje. U većini slučajeva, bilo da radite prozorske ili web-aplikacije (pa čak i aplikacije za mobilne uređaje ili web-servise), koristit ćete neke podatke. Ti će podaci ponekad biti spremljeni u zasebnim datotekama, no najčešće ćete koristiti neku bazu podataka.

Za dohvaćanje podataka i komuniciranje s bazama podataka pod .NET-om postoji posebna tehnologija, nazvana ADO.NET. Ona ima svoju impozantnu prošlost, a prelaskom pod okrilje .NET-a doživjela je tolika poboljšanja da možemo govoriti o potpuno novoj tehnologiji. Microsoft pozicionira ADO.NET (od ActiveX Data Objects) kao primarnu tehnologiju za pristup podacima iz .NET Frameworka, te je stoga njeno poznavanje nužno za bilo kakav rad s podacima. Na svu sreću, iako se radi o moćnoj tehnologiji, relativno ju je lako naučiti.

Da biste se mogli upustiti u rad s bazama podataka, trebat ćete upoznati osnove jezika SQL, standarda za komunikaciju i rad s bazama. Na sljedećih nekoliko stranica dat ćemo vam pregled mogućnosti koje će vam trebati u svakodnevnom radu s bazama.

Osnove jezika SQL

SQL, *Structured Query Language,* standardni je jezik za komunikaciju s bazom podataka kroz tzv. *Database Management System* (DBMS) – sustav koji manipulira svim podacima u bazi. SQL koriste Oracle, Microsoft SQL Server, Microsoft Access, IBM DB2, Sybase i gotovo svi drugi sustavi na tržištu. Poznavanje SQL-a postaje nužno svakom programeru, bilo da se bavi izradom standardnih aplikacija ili je više okrenut Webu.

Ukoliko ste već upoznati s jezikom SQL i iza sebe imate iskustvo u radu s bazama podataka, možete preskočiti narednih nekoliko stranica, jer će na njima biti objašnjene osnovne SQL naredbe.

SQL jezik sastoji se od naredbi za dodavanje, čitanje, izmjenu i brisanje podataka. Pravila jezika odredio je ANSI (*American National Standards Institute*), što znači da je SQL jezik otvoren i ne kontrolira ga niti jedna tvrtka.

SQL nije program ili razvojna okolina poput Visual Studija .NET. SQL je isključivo opisni jezik (opisni jer opisuje akciju koju treba napraviti) i koristi se za komunikaciju s bazom podataka (njenim DBMS-om).

Kako je SQL samo jezik kojim se zadaju naredbe, on nema nikakvo sučelje kojim biste mogli isprobavati njegov učinak na bazu. Za to je zadužen sam program u kojem razvijate bazu podataka. Radite li sa SQL Serverom, otvorite Query Analyzer. Svi primjeri će biti pokazani na bazi Northwind

koja dolazi sa SQL Serverom i drugim Microsoftovim bazama podataka i idealna je za prikazivanje općenitih mogućnosti.

Ukoliko na raspolaganju nemate SQL Server, s Microsoftovih stranica možete besplatno preuzeti MSDE, bazu podataka koja u sebi ima istu arhitekturu kao i SQL Server, no kako je besplatna, nema razvojno okruženje u kojem biste mogli isprobati SQL upite. U tom slučaju možete upite isprobavati pomoću nekog besplatnog alata za rad s MSDE-om ili izvršavati upite iz Query Buildera, koji je standardni dio okoline Visual Studija i bit će objašnjen kasnije u poglavlju.

Iako je SQL složen jezik s mnogim detaljima koje je potrebno naučiti za optimiziranje upita, ova knjiga će vam pokazati četiri glavne SQL naredbe za dohvat, dodavanje, izmjenu i brisanje zapisa iz tablice.

SELECT naredba

SELECT je najsloženija naredba s ogromnim brojem opcija, a glavna joj je namjena dohvaćanje zapisa. Da ne bismo zaglibili previše u teoriju, mogućnosti SELECT naredbe bit će vam prikazane na primjerima. Pogledajte, za početak, najosnovniji oblik te naredbe:

```
SELECT * FROM Products
```

Znate li imalo engleskog jezika, trebali biste znati što ova naredba radi. Ona vraća sve zapise svih polja iz tablice *Products*.

Zvjezdica u gornjoj naredbi ne znači da želite sve zapise iz tablice, već sva polja. Umjesto zvjezdice mogli ste napisati:

```
SELECT ProductName, UnitPrice FROM Products
```

Takva naredba vratila bi sadržaj polja *ProductName* i *UnitPrice* svih zapisa u tablici. Polja koja želite vratiti odvajaju se zarezom (","), a ako želite vratiti sva polja iz neke tablice, koristit ćete zvjezdicu.

Dohvaćene zapise možete i poredati po određenom kriteriju. Ako biste, recimo, proizvode željeli poredati po cijeni tako da je najskuplji na vrhu, napisali biste:

```
SELECT * FROM Products ORDER BY UnitPrice DESC
```

Ključan je ORDER BY dio poslije kojeg navodite polje po kojem želite sortirati i način sortiranja. Ako napišete DESC, tablica će biti sortirana po tom polju silazno odnosno najveće vrijednosti će biti na vrhu. Suprotno od DESC je ASC, a koristi vam za uzlazno sortiranje, da dobijete najmanje vrijednosti na vrhu.

```
SELECT * FROM Products ORDER BY ProductName ASC
```

Tako ste dobili sve zapise poredane abecedno po imenu proizvoda jer je ime tekstualno polje. Prethodni primjer zapise je poredao *brojčano* jer je polje *UnitPrice* decimalnog tipa.

Zapise možete sortirati i po više polja odvojite li uvjete zarezom. Recimo da imate neku tablicu koja sadržava polja Ime i Prezime. Ako želite poredati sve zapise prvo po prezimenu, a onda po imenu, napisat ćete:

```
SELECT * FROM Tablica ORDER BY Prezime ASC, Ime ASC
```

Ako imate više osoba s istim prezimenom, one će tada biti poredane abecedno po imenu.

I još jedan savjet: ASC je *defaultni* način sortiranja te je opcionalan, pa ga ne morate navoditi. Prethodni biste upit mogli napisati i ovako:

```
SELECT * FROM Tablica ORDER BY Prezime, Ime
```

Naravno, ako želite nešto silazno poredati, morat ćete napisati DESC.

Ponekad nećete htjeti vratiti sve zapise neke tablice, već samo određen broj. Za to će vam poslužiti ključna riječ TOP. Ona je posebno korisna, ako je koristite uz ORDER BY i sortiranje zapisa. Pogledajte kako bi izgledao SQL upit kojim biste vratili 10 najskupljih proizvoda:

```
SELECT TOP 10 * FROM Products ORDER BY UnitPrice DESC
```

Ovaj upit zapravo *selektira* sva polja prvih 10 zapisa iz tablice *Products,* i to kad su poredani s najskupljim proizvodom na vrhu.

Možete vratiti i određeni postotak zapisa iz tablice. Recimo, ako biste htjeli vratiti 50% zapisa iz neke tablice, što je korisno kada ne znate ukupan broj zapisa u tablici, napisali biste:

```
SELECT TOP 50 PERCENT * FROM Products
```

Nemojte da vas zbuni ova zvjezdica uz ključnu riječ TOP. Ako ne želite u prethodnom upitu vratiti sva polja iz tablice, već samo neke od njih, sjetite se primjera s početka i navedite željena polja:

```
SELECT TOP 50 PERCENT ProductName, UnitPrice FROM Products
```

Želite li izbrojati koliko zapisa ima u nekoj tablici, iskoristit ćete COUNT(*):

```
SELECT COUNT(*) FROM Products
```

Prethodni SQL upit vratit će vam broj zapisa u tablici *Products*, no naziv polja u tom slučaju neće imati ime. To je dobra prilika da naučite preimenovati polja koja dohvaćate. Samo poslije polja napišite "AS" i njegov novi naziv. U tom slučaju, naravno, ne možete koristiti zvjezdicu za odabir svih polja.

```
SELECT COUNT(*) AS UkupanBroj FROM Products
```

Isto možete primijeniti i na polja koja već imaju imena:

```
SELECT ProductName AS Ime, UnitPrice AS Cijena FROM Products
```

I još samo jedna sitnica: ako biste, recimo, željeli vratiti ID-eve svih dostavljača proizvoda, prvo bi vam pala na pamet naredba:

```
SELECT SupplierID FROM Products
```

No takvom naredbom vraćate vrijednost polja *SupplierID* iz svih zapisa. Što ako ste tim upitom samo htjeli vidjeti koji dostavljači dostavljaju proizvode? Dobili biste ogroman broj zapisa i morali biste ručno kroz sve njih proći i vidjeti čiji se ID-evi ponavljaju. Stoga je vrlo korisna ključna riječ DISTINCT. Ona će vam vratiti samo *različite* vrijednosti nekog polja.

```
SELECT DISTINCT SupplierID FROM Products
```

Tako biste vratili samo ID-eve dostavljača proizvoda bez ponavljanja tih ID-eva.

Postavljanje uvjeta

Kao što ste možda primijetili, gornji primjeri su veoma zgodni za različita vraćanja zapisa. No oni su dostatni samo za neke osnovne manipulacije i ne pružaju vam dovoljne mogućnosti. WHERE ključna riječ služi vam za postavljanje uvjeta koje dohvaćeni zapisi moraju ispunjavati. Ona se koristi i u SELECT naredbi, no i u UPDATE i DELETE naredbama, kao što će biti pokazano kasnije.

Osnovna sintaksa korištenja WHERE uvjeta izgleda ovako:

```
SELECT * FROM Tablica WHERE uvjeti ORDER BY Polje
```

Dijelovi gornjeg upita "SELECT * FROM Tablica" i "ORDER BY Polje" nisu previše važni i služe jedino da vam pokažu u kojem dijelu SQL naredbe se pojavljuje WHERE – poslije FROM u kojem određujete tablicu iz koje vraćate rezultate i prije ORDER BY kojim sortirate rezultate.

Kao uvjet koji zapisi moraju ispunjavati možete postaviti neograničeno mnogo provjera. Kao i u nekim programskim jezicima s Basic sintaksom, možete koristiti operatore manje ("<"), veće (">"), jednako ("="), manje ili jednako ("<="), veće ili jednako (">=") ili pak različito ("<>"). Provjere također možete stavljati u zagrade i odvajati ih logičkim operatorima poput AND, OR, NOT i XOR. Pogledajte primjer:

```
SELECT * FROM Products WHERE SupplierID = 1 AND ProductName = 'Chang'
```

Možete pokušati napraviti i malo složenije uvjete:

```
SELECT * FROM Products WHERE (SupplierID = 1 OR SupplierID = 2) AND NOT ProductName
= 'Chang'
```

Ako radite s datumima, postoji niz predefiniranih funkcija koje vam pritom mogu pomoći. Tako, recimo, trenutni datum i vrijeme vraća funkcija *GetDate()* koju možete iskoristiti u upitu:

```
SELECT * FROM Orders WHERE RequiredDate <= GetDate()
```

Gornji upit vratit će sve narudžbe čija se dostava već trebala obaviti, znači prije sadašnjeg trenutka. Funkcija *Now()* vraća i sekunde pa je preciznost upita veća.

Veoma su korisne i funkcije koje iz nekog datuma vraćaju godinu, mjesec ili dan – *Year()*, *Month()* i *Day()*. Ako biste željeli napisati upit koji će vratiti sve narudžbe načinjene prije 5 godina na današnji datum (ako je danas 7.9.2004, tražite vijesti od 7.9.1999), napisali biste:

```
SELECT * FROM Orders WHERE (Year(OrderDate) = Year(GetDate()) – 5) AND
(Month(OrderDate) = Month(GetDate())) AND (Day(OrderDate) = Day(GetDate()))
```

Gornji upit vraća sve narudžbe kojima je godina naručivanja jednaka trenutnoj godini minus 5 (dakle, prije 5 godina), mjesec im je jednak trenutnom mjesecu, a dan u mjesecu isti kao i danas.

Ako želite uspoređivati tekstualne nizove, poput polja *ProductName*, korištenje operatora za usporedbe uspoređivat će nizove abecedno i pazeći na velika i mala slova. Na svu sreću, na raspolaganju vam stoji LIKE operator. Njega možete smatrati usporedbom jednakosti, no s malo većim mogućnostima.

Korištenjem LIKE operatora ne trebate paziti na velika i mala slova, a možete koristiti i *wildcard* operatore, poput "%" ili "_". Kao što i sama riječ znači, uspoređuje se vrijednost koja *nalikuje na* neku drugu. Pogledajte upit kojim tražite sve proizvode koje imaju "cho" u imenu:

```
SELECT * FROM Products WHERE ProductName LIKE '%cho%'
```

Tako ćete pronaći sve zapise koji bilo gdje u imenu imaju riječ "cho". Pritom ta riječ može biti jedina u polju, a može biti nešto i prije ili poslije nje, zbog upotrebe *wildcarda* "%" koji označava *bilo koje vrijednosti*. Isto tako, može se spominjati i "CHO" ili "cHo", a vaš će upit svejedno vratiti i te zapise, zbog načina rada LIKE operatora.

Drugi *wildcard* operator je "_", i zamjenjuje jedan znak u nizu, za razliku od "%" koji zamjenjuje neodređeni broj znakova (od 0 do neograničeno mnogo).

```
SELECT * FROM Products WHERE ProductName LIKE '%c_o%'
```

Gornji upit vratio bi sve proizvode koji u naslovu imaju riječi poput "cpo", "cho", "cdo" i slično.

Iako postavljanje uvjeta može biti veoma složeno, ovdje prikazani upiti trebali bi vam biti dovoljni za osnovni rad s bazom podataka i njeno korištenje u aplikacijama. Naravno, ukoliko vam na pamet padne neka složenija ideja, nemojte se ustručavati pogledati u *Books Online* koji dolaze uz SQL Server ili potražiti na Webu.

INSERT naredba

Osnovna namjena INSERT naredbe je dodavanje novih zapisa u neku tablicu. Pogledajte kako izgleda njena osnovna sintaksa:

```
INSERT INTO Tablica (popis_polja) VALUES (popis_vrijednosti)
```

U gornjem upitu *popis_polja* označuje polja u koje želite ubaciti neke vrijednosti odvojene zarezom, a *popis_vrijednosti* vrijednosti za koje želite da poprime ta polja. Ako biste željeli ubaciti novu vijest u tablicu *Vijesti* napisali biste:

```
INSERT INTO Products (ProductName, SupplierID, CategoryID, QuantityPerUnit,
UnitPrice, UnitsInStock, UnitsOnOrder, ReorderLevel, Discontinued) VALUES ('Moj
novi proizvod', 1, 3, '10 komada', 15.0000, 30, 0, 10, 0)
```

Za razliku od SELECT naredbe koja vraća zapise koje ste zatražili, INSERT naredba ne vraća ništa, već samo izvršava dodavanje zapisa. Primijetite da u gornjem upitu niste specificirali vrijednost polja ProductID, jer je ono tipa *identity,* pa se za njega brine sama baza i automatski ga dodaje.

I još samo kratka napomena: sintaksa INSERT naredbe veoma je jednostavna. Ubacujete li tekstualne nizove, stavite ih u polunavodnike. Ubacujete li pak brojčane ili logičke vrijednosti, ne trebate koristiti polunavodnike. Kod korištenja INSERT naredbe jedino morate paziti da vrijednosti koje ste napisali unutar VALUES dijela prate poredak polja – tj. u prvo polje ubacit će se prva vrijednost, u drugo polje druga vrijednost itd.

UPDATE naredba

UPDATE naredba izmjenjuje vrijednosti već postojećih zapisa u tablici. Njena osnovna sintaksa je sljedeća:

```
UPDATE Tablice SET Polje1 = Vrijednost1, Polje2 = Vrijednost2, ... WHERE uvjet
```

Primijetite da obavezno morate navesti uvjet koji zapisi moraju ispuniti da bi njihova vrijednost bila promijenjena. U slučaju da ne napišete uvjet i izostavite cijeli WHERE dio, izmijenit će se svi zapisi u tablici, što će vam vjerojatno donijeti više štete nego koristi.

Pri postavljanju uvjeta u UPDATE naredbi koristit će vam ID polje. Evo kako biste promijenili ime i cijenu proizvoda s ID-em 1 (vrijednosti ostalih polja ostale bi neizmijenjene):

```
UPDATE Products SET ProductName = 'Moj proizvod', UnitPrice = 10.5000 WHERE
ProductID = 1
```

Isto kao INSERT naredba, UPDATE ne vraća nikakvu vrijednost, već samo izmjenjuje vrijednosti zapisa. U slučaju da nije pronađen niti jedan zapis koji odgovara uvjetima, ništa se neće izmijeniti, no vi nećete dobiti poruku o tome. SQL upit će se izvršiti, a na vama je da provjerite što je on zapravo napravio.

DELETE naredba

Vjerovali ili ne, DELETE naredba briše zapise iz neke tablice. Njena osnovna sintaksa je:

```
DELETE FROM Tablica WHERE uvjet
```

Primijetite da ovdje ne morate navesti koja polja brišete jer ona briše cijele zapise. Zato je bitan *uvjet* koji ima isti oblik kao i kod SELECT i UPDATE naredbi. Evo kako biste izbrisali proizvod s ID-em 79:

```
DELETE FROM Products WHERE ProductID = 79
```

No prije nego što se upustite u brisanje zapisa, imajte na umu da je to nepovratna akcija. Ne postoji nikakav *Undo* ili nešto slično čime biste mogli vratiti izbrisane podatke.

Spajanje više tablica

SQL vam omogućava spajanje povezanih tablica i dohvaćanje njihovih vrijednosti jednim upitom. Naprimjer, ako biste željeli spojiti tablice *Products* i *Suppliers*, koje su i logički povezane jer uz svaki proizvod postoji informacija o njegovu dostavljaču, napisali biste:

```
SELECT * FROM Products INNER JOIN Suppliers ON Products.SupplierID =
Suppliers.SupplierID
```

Gornji upit dohvaća tablicu *Products* i dodaje joj tablicu *Suppliers* tako da ih spaja po poljima *SupplierID*. Primjerice, ako se u nekom retku tablica *Products* pojavljuje *SupplierID* 1, tom retku će biti dodan redak iz tablice *Suppliers* u kojem je *SupplierID* jednak 1. Tako uz svaki proizvod imate informacije o dostavljaču koji ju je dostavio. Rezultat spajanja tih tablica bit će nova tablica, koja će u sebi imati sljedeća polja:

```
ProductID, ProductName, SupplierID, CategoryID, QuantityPerUnit, UnitPrice,
UnitsInStock, UnitsOnOrder, ReorderLevel, Discontinued, SupplierID, CompanyName,
ContactName, ContactTitle, Address, City, Region, PostalCode, Country, Phone, Fax,
HomePage
```

Kao što vidite, radi se o zbroju svih polja iz obje tablice. Prvo su dohvaćena polja iz prve tablice, tj. iz tablice *Products*, a njima su dodana sva polja iz tablice *Suppliers*. Zbog toga su i neki podaci redundantni odnosno ponavljaju se. Tako se ponavlja ID dostavljača u poljima *SupplierID*, jer su ona trebala biti jednaka za spajanje tablica.

Zato je mnogo prihvatljivije točno odrediti koja polja želite iz spojene tablice. Evo kako biste to napravili, uzevši u obzir samo osnovne informacije o proizvodima i njihovim dostavljačima:

```
SELECT Products.ProductID, Products.ProductName, Products.UnitPrice,
Suppliers.CompanyName FROM Products INNER JOIN Suppliers ON Products.SupplierID =
Suppliers.SupplierID
```

Rezultat toga upita prikazan je u tablici 9-1. Dohvatili ste samo polja koja će vam trebati u programu i tako ubrzali cijeli postupak izbacivši ona nepotrebna. Također, samo ćete prikazivati ime i cijenu proizvoda te ime tvrtke dostavljača, a ostalo vam nije potrebno.

Pri navođenju polja iz spojene tablice koje želite u konačnom rezultatu, morali ste navesti i tablicu u kojoj se nalaze. Isto možete koristiti i u jednostavnijim upitima. Naravno, ne morate, jer ako je u igri samo jedna tablica, podrazumijeva se otkud dolaze polja, ali samo da uočite sintaksu:

```
SELECT Products.ProductName, Products.UnitPrice FROM Products
```

Tablica 9-1:
Kompliciranijim upitom dohvaćena su samo neka polja iz spojenih tablica (u tablici je prikazan samo izvadak dohvaćene tablice, i to zapisi s ID-evima 7, 8 i 9).

ProductID	ProductName	UnitPrice	CompanyName
7	Uncle Bob's Organic Dried Pears	30.0000	Grandma Kelly's Homestead
8	Northwoods Cranberry Sauce	40.0000	Grandma Kelly's Homestead
9	Mishi Kobe Niku	97.0000	Tokyo Traders

Spajanje tablica bit će vam veoma korisno tamo gdje je potrebno u što manje upita dohvatiti što više podataka. Ne možete si dopustiti komfor i upućivati gomile SQL upita – što više korisnika koristi vašu aplikaciju, baza će biti sve opterećenija i može doći do zagušenja. Spajanje relacijskih tablica zapravo je jedino rješenje tog problema, a omogućava vam pisanje mnogo urednijeg kôda.

Nakon što ste se upoznali s osnovama jezika SQL, slijedi prava stvar. U nastavku ćete se upoznati s osnovnim komponentama ADO.NET-a i naučiti kako iz .NET aplikacija raditi sa stvarnim podacima iz baza.

Komponente ADO.NET-a

Ukoliko već imate iskustva s programiranjem aplikacija koje pristupaju podacima, zasigurno ste upoznali preteču ADO.NET-a, naziva ADO. Središnji dio ADO arhitekture bio je *recordset* – radilo se o reprezentaciji podataka sličnoj nekoj tablici po kojoj ste se mogli kretati u svojoj aplikaciji, dohvaćati određene retke, stupce i slično. ADO.NET donosi niz novih mogućnosti, kao što su komponente *DataReader* i *DataSet* koje u potpunosti zamjenjuju *recordset* te on više ne postoji – to je prva prepreka koju morate svladati želite li kvalitetno iskoristiti mogućnosti ADO.NET-a.

Na *DataSet* se može gledati kao na složeniju verziju *recordseta* jer može u sebi sadržavati kompletnu reprezentaciju baze podataka; primjerice, više tablica, njihove relacije, primarne ključeve, *poglede* (*viewove*), omogućava sortiranje, filtriranje itd.

No najveća razlika između *DataSeta* i *recordseta* je u načinu njihova rada – dok je *recordset* za vrijeme svog korištenja stalno bio spojen na bazu podataka i tako konstantno trošio resurse (pogotovo ako bi ga programer zaboravio zatvoriti pa bi ostao u memoriji i nakon završetka programa), *DataSet* nije spojen na bazu i u sebi sadržava samo podatke koji su u njega stavljeni. Podatke u *DataSetu* možete u svojoj aplikaciji mijenjati te, kad ste završili, sve spremiti natrag u bazu. Više nema potrebe da veza s bazom bude stalno otvorena, što uvelike poboljšava performanse sustava.

Sve opisane komponente bit će kasnije detaljnije razrađene, a zasad opisi služe samo da si stvorite predodžbu i dobijete sliku mogućnosti ADO.NET-a.

Uz *DataSet* usko je vezana i *DataAdapter* komponenta. Ona predstavlja vezu na bazu podataka i zadužena je prvenstveno za popunjavanje *DataSeta*. Ona će i sve promjene načinjene u *DataSetu* spremiti natrag u bazu podataka.

Na *DataReader* može se gledati kao na inačicu *recordseta* koja je sposobna isključivo prikazivati podatke iz baze podataka (tzv. *read-only* odnosno podaci se ne mogu mijenjati u *DataReaderu* te potom spremiti natrag u bazu) i kretati se kroz dohvaćene zapise samo unaprijed (tzv. *forward-only* odnosno omogućava slijedno čitanje dohvaćenih zapisa bez mogućnosti vraćanja unatrag za jedan zapis – želite li se vratiti unatrag, jedino što vam se nudi je kretanje ispočetka). Unatoč svim navedenim nedostacima, *DataReader* može biti izvrstan izbor u većini slučajeva, pogotovo kad budete razvijali ASP.NET aplikacije za koje je važan brz pristup podacima.

Da biste dobili dojam odnosa, zgodno je napomenuti da *DataAdapter* za popunjavanje *DataSet* objekata interno koristi *DataReader* objekte.

Arhitektura

ADO.NET je, ponovimo, najnovija verzija Microsoftove tehnologije za univerzalan pristup podacima. To znači da je ADO.NET potpuno neovisan o izvoru podataka. On se temelji na XML-u, te je stoga potpuno svejedno odakle se podaci dovlače jer se oni interno spremaju u XML-u. Nakon što napišete programsku podršku i ostvarite funkcionalnost povezivanja i rada s bazom podataka, uz samo male promjene parametara možete raditi i sa sasvim drugim izvorom podataka.

To mogu biti baze smještene na SQL Serveru, poslužitelju Oracle, obične Access baze ili jednostavno XML datoteke – s tim podacima radite na gotovo identičan način, što predstavlja izuzetan dobitak, jer jednom kad svladate osnovne metode ADO.NET-a, svi ti izvori podataka vam stoje na raspolaganju.

Zahvaljujući nasljeđivanju, koje je osnova biblioteke .NET klasa, i ADO.NET klase nasljeđuju baznu funkcionalnost iz osnovnih klasa, što bitno pojednostavljuje programiranje. U tablici 9-1 navedeni su *namespaceovi* u kojima je pohranjena ADO.NET funkcionalnost.

Tablica 9-2:
Namespaceovi koje koristi ADO.NET

Namespace	Opis
System.Data	sadržava osnovne objekte, kao što su *DataTable*, *DataColumn*, *DataView*; to je temeljni namespace iz kojeg su izvedeni drugi
System.Data.Common	definira generičke objekte koje dijele svi *provideri* za pristup podacima, kao što su *DataAdapter*, *DataColumnMapping* ili *DataTableMapping*; u većini slučajeva ovaj *namespace* nećete sami koristiti (jer se on koristi interno), već samo ako stvarate svoj provider za pristup podacima
System.Data.OleDb	*provider* za spajanje na sve OLE DB izvore podataka; postiže mnogo bolje performanse nego ODBC *provider*
System.Data.SqlClient	*provider* za spajanje isključivo na SQL Server verzija 7.0 ili novijih; ukoliko, dakle, koristite SQL Server, korištenjem ovog providera postići ćete daleko bolje performanse nego korištenjem OLE DB *providera*, jer on interno direktno koristi SQL Server API
System.Data.SqlTypes	klase za rad sa SQL Serverima tipovima podataka
System.Data.Odbc	*provider* za spajanje na sve ODBC izvore podataka – dostupan je kao besplatan dodatak, a može se preuzeti s Microsoftovih web-stranica

Želite li se s .NET-om spajati na Oracle bazu podataka, na raspolaganju vam stoji ".NET Managed Provider for Oracle", besplatan dodatak koji se može preuzeti s Microsoftovih web-stranica. Njegovom instalacijom dobivate System.Data.OracleClient *namespace* u kojem se nalaze svi potrebni objekti za rad s Oracle bazama.

U nastavku ćemo objasniti najčešće korištene objekte ADO.NET-a – *Command*, *Connection*, *DataReader* i *DataAdapter*. Oni su definirani u svim klasama za pristup podacima, od *System.Data.OleDb* i *System.Data.Odbc* do *System.Data.SqlClient*.

Kao što vidite, za pristup podacima iz .NET-a ključno je odabrati *provider*. On, dakako, direktno ovisi o izvoru podataka – ukoliko koristite SQL Server, upotrijebit ćete *System.Data.SqlClient*. Ukoliko pak koristite neku drugu bazu podataka, upotrijebit ćete *System.Data.OleDb*, ako za tu bazu podataka imate dostupne *drivere* OLE DB, ili *System.Data.Odbc* – ako imate dostupne *drivere* ODBC.

Objekt *Connection*

Prvi korak u izradi programa temeljenog na nekom izvoru podataka je povezivanje s tim izvorom. Za to služi objekt *Connection*. Spajanje na bazu njegovim korištenjem uistinu je jednostavno i svodi se na definiranje tzv. *connection stringa* odnosno upute u kojoj su definirani izvor podataka, način spajanja i druge postavke.

U nastavku ćemo objasniti osnovne dijelove *connection stringa*. U većini slučajeva nećete se morati zamarati njihovim postavljanjem, već ćete *connection string* stvoriti direktno iz Visual Studio .NET okoline, kao što će biti pokazano kasnije u poglavlju.

Tablica 9-3:
Dijelovi connection stringa za spajanje na SQL Server bazu podataka (nije ih potrebno sve uvijek navoditi)

Naziv	Opis
Connection Timeout	vrijeme tijekom kojeg se pokušava spojiti na bazu podataka – ukoliko vrijeme istekne i veza se ne ostvari, podiže se iznimka; po *defaultu* je to vrijeme postavljeno na 15 sekundi
Data Source	ime ili IP adresa poslužitelja na kojem je smještena baza podataka
Initial Catalog	ime baze podataka koja se otvara
Integrated Security	određuje da li će se pri spajanju na server koristiti postavke korisničkog računa koji otvara konekciju (*true*) ili će se definirati korisničko ime i zaporka pod kojim će se ostvariti veza (*false*)
Password	zaporka za spajanje na bazu podataka; ukoliko se koristi *integrated security*, ne treba je navoditi
Persist Security Info	kad se postavi na *false*, svi kritični podaci za sigurnost (npr. zaporka) ne vraćaju se zajedno s otvorenom konekcijom
User ID	korisničko ime za spajanje na bazu podataka; ukoliko se koristi *integrated security*, ne treba ga navoditi

U tablici 8-2 navedeni su svi dijelovi *connection stringa* za spajanje na SQL Server bazu podataka. No najčešći dio *connection stringa* je *Provider* koji određuje na koji se tip baze podataka spaja. Njega ćete koristiti uvijek, osim kad se spajate na SQL Server, jer se pritom podrazumijeva da se radi o SQL Server bazi podataka.

Kombinirajući sve dijelove *connection stringa* vrlo lako možemo napraviti izraz pomoću kojeg biste se spojili na neku bazu podataka:

```
Data Source=imeServera;Initial Catalog=northwind;Integrated Security=true;
```

Prethodnim biste se *connection stringom* spojili na *northwind* bazu podataka smještenu na serveru imena *imeServera*. Za spajanje na bazu podataka koristili bi se podaci korisnika koji pokreće aplikaciju.

Objekt *Command*

Pojednostavljeno rečeno, objekt *Command* služi za izvršavanje različitih upita direktno na bazi podataka. To mogu biti obični SQL upiti ili pozivi spremljenih procedura, a unutar samih poziva mogu se definirati i različiti parametri koji se koriste za prosljeđivanje vrijednosti upitima.

Svi objekti koji će biti objašnjeni u ovom dijelu postoje i za OLE DB *provider*, i za ODBC *provider* i za SQL *provider*. Primjerice, objekt *Command* će se u svojoj OLE DB verziji zvati *OleDbCommand*, u ODBC verziji će se zvati *OdbcCommand*, a u SQL Server verziji *SqlCommand*. S njima se radi na potpuno isti način, a razlika je samo u njihovu internom načinu rada, što vas kao programera aplikacija u .NET-u ne treba zamarati.

Pogledajmo na primjeru kako bi izgledalo otvaranje veze prema bazi podataka te stvaranje *Command* objekta.

```
string connStr = "Data Source=mojServer;Initial Catalog=northwind;Integrated
      Security=true;";
string sqlUpit = "SELECT * FROM Orders";

SqlConnection conn = new SqlConnection(connStr);
```

```
conn.Open();
SqlCommand cmd = new SqlCommand(sqlUpit, conn);
```

U našem primjeru radimo sa SQL Serverom i, kao što vidite, za stvaranje veze prema bazi podataka koristi se objekt *SqlConnection*. Pri njegovu stvaranju za parametar se prosljeđuje *connection string* na temelju kojeg ADO.NET zna točno koju bazu treba otvoriti.

Da bi prethodni primjer radio, potrebno je dodati nazive korištenih *namespaceova* na početak programa. Kako koristimo objekte za rad sa SQL Serverom, na početku programa trebali bismo dodati:

```
using System.Data.SqlClient;
```

Prethodni primjer za naredbu bazi podataka koristi običan SQL upit. Objekt *Command* pak može puno pomoći pri izvršavanju spremljenih procedura na SQL Serveru. Recimo da imamo proceduru sljedećeg sadržaja:

```
CREATE PROCEDURE getOrder
 @OID int
AS
 SELECT * FROM Orders WHERE OrderID = @OID
GO
```

Spremljena procedura ili *stored procedure* je SQL naredba (ili više njih) spremljena na SQL Serveru. Kako je interno pohranjena, njeno izvršavanje je optimizirano i izvršava se mnogo brže od običnih SQL upita koji se postavljaju na bazu jer je poslužitelj ne treba kompajlirati i već mu je poznat plan njena izvršavanja.

Radi se o jednostavnoj spremljenoj proceduri koja prima jedan parametar ID narudžbe i na temelju njega vraća odgovarajući zapis. Sad trebamo pozivu procedure dodati i jedan parametar, a to ćemo obaviti zahvaljujući objektu *SqlParameter*. No prije toga moramo dodati još jednu liniju u kojoj ćemo obavijestiti ADO.NET da ne izvršavamo direktno neki SQL upit, već da pozivamo spremljenu proceduru. Pretpostavimo da već postoji otvorena veza s bazom podataka u objektu *conn*.

U primjerima nećemo i pozivati objekt *Command* i dohvaćati njegove rezultate jer će nam za to biti potrebno razumijevanje *DataReadera* i drugih objekata, što ćemo obraditi kasnije. Zasad je jedino bitno njihovo stvaranje i namještanje parametara.

```
string sqlUpit = "getOrder";

SqlCommand cmd = new SqlCommand(sqlUpit, conn);
cmd.CommandType = CommandType.StoredProcedure;
```

Koristili smo *CommandType* svojstvo u koje smo postavili konstantu i istoimenog objekta. Tako ADO.NET zna da se radi o pozivu spremljene procedure jer sam SQL upit ne bi bio ispravan – radi se samo o nazivu spremljene procedure.

Kod pozivanja spremljenih procedura postoje ulazni i izlazni parametri. Ulazni parametri su oni koji se prosljeđuju spremljenoj proceduri i koje ona koristi za obavljanje određenih akcija, dok su izlazni parametri oni koje spremljena procedura vraća programu kao rezultat njihovih akcija. Izlazni parametri su najčešće obične vrijednosti varijabli; izlaznim parametrima ne smatraju se zapisi iz tablica.

Prethodni primjer ne bi bio ispravan jer spremljena procedura *getOrder* koristi jedan ulazni parametar. U programu trebamo zato stvoriti parametar i proslijediti ga spremljenoj proceduri. Za to ćemo iskoristiti spomenuti objekt *SqlParameter*.

```
SqlParameter parametar = cmd.Parameters.Add(new SqlParameter("@OID", SqlDbType.Int,
4));
```

Primijetite da objektu *cmd* koji sadrži komandu bazi podataka dodajemo parametar u kolekciji *Parameters*. Radi se o novom objektu koji ima ime @OID i tipa je *Int*. Kao što ćete vidjeti kasnije, najčešće nećete trebati uopće paziti kojeg je tipa parametar koji stvarate, jer ćete sve obaviti iz radne okoline Visual Studija, koji će ga sam namjestiti na odgovarajuću vrijednost.

Kao što smo rekli, parametri mogu biti ulazni i izlazni. Smjer samog parametra možete namjestiti na način prikazan u sljedećem primjeru, a u tablici 9-3 možete vidjeti sve moguće *smjerove* parametara.

```
parametar.Direction = ParameterDirection.Input;
```

Smjer	Opis
Input	ulazni parametar čija se vrijednost prosljeđuje spremljenoj proceduri ili SQL upitu
Output	izlazni parametar koji je i u spremljenoj proceduri definiran kao "OUTPUT" i čija se vrijednost vraća natrag u program
InputOutput	parametar koji može služiti i kao ulazni i kao izlazni parametar, što znači da se njegova vrijednost šalje spremljenoj proceduri, ali je njegova nova vrijednost dostupna nakon izvršavanja procedure
ReturnValue	izlazni parametar koji vraća procedura; može se imati samo jedan za spremljenu proceduru

Tablica 9-4: ***Moguća svojstva ParameterDirection kolekcije koja određuju smjer parametra***

I na samom kraju, ukoliko koristite ulazni parametar, morate postaviti njegovu vrijednost. To radite tako da mu namjestite svojstvo *Value*:

```
parametar.Value = 50;
```

Ukoliko pak imate izlazni parametar, nakon pozivanja spremljene procedure ili SQL upita, njegovu vrijednost možete iščitati:

```
string rezultat = cmd.Parameters("@IzlazniParametar").Value;
```

Objekt *DataReader*

DataReader je, kao što smo spomenuli na početku teksta, jednostavan optimiziran objekt, prilagođen isključivo čitanju zapisa prema naprijed (što znači da možete ići samo zapis po zapis unaprijed, bez mogućnosti vraćanja unatrag). To bi vam u većini slučajeva trebalo biti dovoljno, ako želite samo dohvatiti određene zapise iz baze, nad njima napraviti određene akcije i ispisati u svojoj aplikaciji.

Povratne vrijednosti, gdje ste?

Spremljene procedure u SQL Serveru mogu vraćati vrijednosti s naredbom RETURN. Pogledajte jednostavnu proceduru u sljedećem primjeru (Napomena: radi se o izmišljenim podacima iz izmišljene tablice, a procedura dohvaća identifikator zapisa zadnje narudžbe.):

```
CREATE PROCEDURE GetLastOrderID
AS
  DECLARE @LastID INT
  SET @LastID = (SELECT TOP 1
    OrderID FROM Orders ORDER BY
    OrderDate DESC)
  RETURN @LastID
GO
```

Da biste dohvatili vrijednost koju vraća spremljena procedura, trebate stvoriti parametar imena "RETURN VALUE" i namjestiti mu smjer na *ReturnValue*:

```
SqlParameter param =
  cmd.Parameters.Add(new
  SqlParameter("RETURN VALUE",
  SqlDbType.Int, 4));
param.Direction =
  ParameterDirection.ReturnValue;
```

Prethodni primjeri, dakako, rade samo na SQL Serveru.

Evo kako biste napunili *DataReader* objekt i pročitali sve dohvaćene zapise. Pretpostavimo da već postoji objekt *Command* koji dohvaća određene zapise (recimo, najobičnija "SELECT * FROM tablica" naredba).

```
SqlDataReader dr = cmd.ExecuteReader();
while (dr.Read())
{
 // čitanje pojedinih zapisa
}
```

Ovdje smo prvi put pokrenuli objekt *Command* – izvršili smo naredbu *ExecuteReader()* koja dohvaća podatke prilagođene *DataReader* objektu i koje možemo direktno u njega učitati.

Način rada je veoma sličan *recordset* objektu iz starog ADO-a: u *while* petlji čitamo sve zapise iz *DataReadera*. Ona će se izvršavati sve dok ima zapisa u *DataReaderu* jer se naredbom *Read()* automatski pozicioniramo na naredni zapis.

Želite li dohvatiti vrijednost nekog stupca pojedinog retka, možete koristiti indeks ili ime tog stupca. Primjerice, ukoliko ste izvršili naredbu "SELECT ime, prezime FROM osobe", na raspolaganju imate dva stupca. Njih možete dohvatiti indeksima 0 i 1 ili imenima "ime" i "prezime".

```
string vrijednost;
while (dr.Read())
{
 vrijednost = dr[0].ToString();
 vrijednost = dr["ime"].ToString();
}
```

Objekt *DataSet*

Na *DataSet* možete gledati kao na kopiju podataka iz baze podataka spremljenu u memoriji radi bržeg pristupa. To može biti samo jedna tablica ili više tablica, a one čak mogu biti dohvaćene iz različitih baza podataka. Radi se, dakle, o kompletnoj kopiji podataka koja je u potpunosti odvojena od svog izvora i potpuno neovisna o njemu.

Radeći s *DataSet* objektom ne komunicirate direktno s bazom podataka. Tek nakon što obavite sve operacije nad *DataSetom* i njegovim podacima, možete te podatke poslati natrag na njihov izvor na obradu, što može uključivati mijenjanje originalnih podataka u bazi, brisanje, ili dodavanje novih.

Objekt *DataSet* nije ograničen samo na rad s bazama podataka, već može učitavati i spremati podatke i iz XML datoteka, s kojima se radi jednako kao s bazama podataka.

Objekt *DataTable*

Svaki objekt *DataSet* sadržava kolekciju objekata *DataTable*. Kao što i samo ime kaže, radi se o kolekciji tablica spremljenih u *DataSetu*, a to može biti jedna ili više njih. Na objekte *DataTable* može se gledati kao na tablice u nekoj bazi podataka ili u drugom izvoru podataka.

Kako je cilj razvojnog tima ADO.NET-a bio učiniti objekt DataSet ključnim za rad s podacima u .NET-u, dodane su mu mnoge korisne mogućnosti. Tako tablicama (objektima *DataTable*) možete dodavati primarne ključeve (*PrimaryKeys*) ili ih međusobno povezivati korištenjem *DataRelations* kolekcije. S tim mogućnostima objekt *DataSet* doista postaje punopravna kopija sadržaja baze, spremljena u memoriji.

Objekte *DataTable* možete stvarati programski – stvorite novu tablicu, dodate joj stupce, dodate joj retke sadržaja i slično – ili ih puniti iz baze podataka, što je ipak češći način rada. U nastavku slijedi primjer punjenja *DataSeta* odnosno jedne njegove tablice sadržajem iz baze podataka.

Za to ćemo iskoristiti *DataAdapter* – primjer će dohvatiti sve podatke iz neke tablice i proslijediti ih nekoj kontroli koja može prikazivati podatke, primjerice *Repeater* kontroli u ASP.NET aplikacijama (sljedeći primjer će biti detaljnije objašnjen u narednom poglavlju).

```
SqlConnection sqlConn = new
  SqlConnection("server=(local);database=Northwind;Integrated Security=true");

SqlDataAdapter sqlComm = new SqlDataAdapter("SELECT * FROM Orders", sqlConn);

DataSet ds = new DataSet();
sqlComm.Fill(ds);

Kontrola.DataSource = ds;
Kontrola.DataBind();
```

Ključna naredba je *sqlComm.Fill(ds)* kojom se puni *DataSet* podacima iz baze koji se dohvaćaju preko *DataAdaptera*. Nakon toga, nekoj kontroli za izvor podataka postavljamo stvoreni *DataSet*.

Svakoj tablici u *DataSet* objektu pristupate preko kolekcije *Tables*. Pritom možete koristiti indeks tablice ili njeno ime, primjerice:

```
ds.Tables[0]
ds.Tables["Orders"]
```

Objekt *DataColumn*

Kao što i svaka tablica u bazi podataka ima svoje stupce s odgovarajućim svojstvima, poput tipa podataka koji sadržava, duljine i slično, tako i objekt *DataSet* odnosno svaki njegov objekt *DataTable* sadržava niz stupaca odnosno objekata *DataColumn*.

Stupci nekog objekta *DataTable* predstavljaju njegovu strukturu, dakle bez podataka. Svaki stupac tako ima niz atributa koji utječu na način njegova rada. Primjerice, ukoliko je svojstvo *AllowDBNull* postavljeno na *true*, stupac će moći primati i *null* vrijednosti. Tip podataka nekog stupca sprema se u atribut *DataType*, a ukoliko želite onemogućiti promjene sadržaja tog stupca, postavit ćete svojstvo *ReadOnly* na *true*.

Kao što je već rečeno, objekte *DataTable* možete puniti iz baza podataka ili ih direktno stvarati. U narednom primjeru bit će prikazano stvaranje tablice s dva stupca i njeno dodavanje u objekt *DataSet*.

```
DataColumn Stupac1 = new DataColumn();
DataColumn Stupac2 = new DataColumn();

Stupac1.DataType = System.Type.GetType("System.Int32");
Stupac1.ColumnName = "OsobaID";
Stupac1.AutoIncrement = true;
Stupac1.ReadOnly = true;

Stupac2.DataType = System.Type.GetType("System.String");
Stupac2.ColumnName = "OsobaImePrezima";

DataTable Tablica = new DataTable("Osobe");
Tablica.Columns.Add(Stupac1);
Tablica.Columns.Add(Stupac2);

DataSet ds = new DataSet();
ds.Tables.Add(Tablica);
```

Dakle, cilj nam je bio stvoriti tablicu koja će služiti pohranjivanju informacija o osobama te će imati dva stupca – prvi će služiti spremanju jedinstvenog identifikatora osobe, a drugi je za njezino ime i prezime.

Zato smo na početku stvorili dva stupca – prvi je bio tipa *Int32*, ime smo mu postavili na *OsobaID*, postavili smo mu i svojstvo *AutoIncrement* na *true*, što znači da će se identifikator osoba automatski povećavati kako budemo dodavali nove osobe i mi ga nećemo morati mijenjati, te smo stoga i zabranili njegovo mijenjanje postavljenjem svojstva *ReadOnly* na *true*.

Na sličan način smo namjestili i svojstva drugog stupca – tip podataka koji će sadržavati je *string*, jer je predviđeno da u njega upisujemo ime i prezime osobe, a shodno tome smo mu postavili i ime.

Sljedeći korak bio je stvaranje nove tablice, tj. objekta *DataTable*. Novu tablicu smo nazvali *Osobe*, a dodali smo joj dva već prije stvorena stupca. Na samom smo kraju stvorili objekt *DataSet* kojem smo dodali netom stvorenu tablicu.

Objekt DataRow

Drugi glavni dio objekta *DataTable*, uz spomenuti *DataColumn*, jest objekt *DataRow*. Radi se pojedinim zapisima u tablici podataka, a oni služe za mijenjanje, dodavanje i brisanje podataka.

Verzije i greške u redcima

Svaki *DataRow*, osim podataka, sadržava i informacije o greškama u retku i njegovim verzijama. Verzije sadržaja nekog retka mogu se pratiti pomoću *RowState* svojstva i *GetChanges* i *HasChanges* metoda pa tako redak može biti u stanju *Added* (redak je dodan u kolekciju), *Deleted* (pozvana je metoda za brisanje), *Detached* (redak nije dio kolekcije), *Modified* (njegov sadržaj je promijenjen) ili *Unchanged* (sadržaj je nepromijenjen). Želite li spremiti sve promjene načinjene u retku, pozvat ćete metodu *AcceptChanges*. Imajte na umu da se pozivom te metode ne mijenjaju podaci originala odnosno u samoj bazi podataka ili koji ste već izvor koristili – tek se pozivom *Update* metode *DataAdaptera* te promjene spremaju, i to samo one u redcima nad kojima je pozvana metoda *AcceptChanges*.

Želite li pak saznati više o greškama koje su se pojavile u retku odnosno njegovu sadržaju, provjerit ćete svojstvo *HasErrors*. Ukoliko ono ima vrijednost *true*, možete pozvati metodu *GetColumnError* da dobijete jedan stupac s greškom ili *GetColumnsInError* da dobijete kolekciju stupaca s greškom.

DataTable, dakle, sadržava kolekciju redaka odnosno kolekciju objekata *DataRow*. S tom kolekcijom se radi kao i sa svakom drugom – želite li dodati novi redak, iskoristit ćete metodu *Add*, a želite li pristupiti pojedinom retku iz kolekcije, iskoristit ćete svojstvo *Item*.

Razlike između *DataReadera* i *DataSeta*

Kao što je već više puta naglašeno, svi podaci koji se koriste kroz ADO.NET nisu konstantno spojeni na svoj izvor, tj. nazivamo ih *disconnected*. Pojednostavljeno rečeno, pristup podacima može se svesti na dva modela – korištenjem *DataReadera* i korištenjem *DataSeta*.

Pri korištenju *DataSeta* podaci se učitavaju u lokalni spremnik, zatim čitaju i mijenjaju te na kraju sinkroniziraju s izvorom podataka. Veza s bazom podataka otvara se na početku, *DataSet* se puni (tj. njegov *DataTable*) i aplikacija može s tom kopijom podataka raditi što želi, a veza s bazom se zatvara. Resursi baze podataka se tako ne troše i ona je slobodna obavljati druge poslove.

DataReader pak ne dopušta mijenjanje tih podataka ili višekratno korištenje – podaci se čitaju samo jednom i već se pri čitanju narednog retka svi prošli podaci brišu iz memorije. No za razliku od *DataSeta*, pri korištenju *DataReadera* je stalno otvorena veza s bazom podataka. Ukoliko bi vaša aplikacija dopuštala korisniku korištenje *DataReader* objekta, to bi moglo predstavljati potencijalan problem – baza podataka vraća redak po redak te čeka na sljedeću naredbu *DataReadera*. Sto-

ga je *DataReader* najbolje koristiti u situacijama kad vam je potreban brz pristup nekim podacima i kad ih sve iščitavate u nekoj petlji, bez interakcije korisnika, te kad ih ne planirate više puta upotrebljavati.

Objekt *DataView*

I na samom kraju, ostao nam je još jedino objekt *DataView*. Njegova namjena je da povezuje podatke s formularima i kontrolama u aplikaciji, a može se koristiti i za pretraživanje, filtriranje, sortiranje i uređivanje podataka.

DataView se temelji na objektu *DataTable* i služi kao poveznica između njegovih podataka i polja predviđenih podatke na formularu, bilo u web-aplikaciji ili prozorskoj aplikaciji. Kao što i samo ime objekta kaže, on nudi stvaranje drugačijih *pogleda* na podatke i njihovo prilagođavanje. Primjerice, možete stvoriti jedan *DataView* koji će prikazivati sve zapise iz objekta *DataTable* koji imaju cijenu veću od 10 i drugi *DataView* koji će prikazivati sve s manjom cijenom zahvaljujući mogućnosti filtriranja redaka.

Primijetite da se pri korištenju objekta *DataView* podaci dohvaćaju samo jednom, i to u *DataTable* objekt, a kasnije se u memoriji filtriraju i obrađuju u drugim *DataView* objektima, što uvelike olakšava posao bazi podataka koja podatke dohvaća samo jednom.

Rad s ADO.NET-om

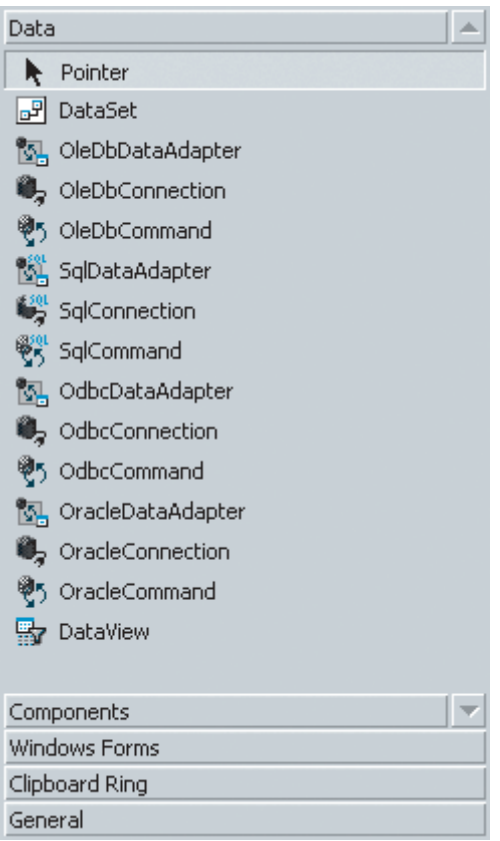
U nastavku ćemo objasniti kako lako napraviti aplikaciju koja radi s podacima. S nekoliko klikova mišem stvorit ćemo vezu s bazom podataka i prikazivati podatke u *DataGrid* kontroli te uz malo kôda te podatke i uređivati.

Za početak, stvorite običnu prozorsku aplikaciju, kao što ste naučili u prethodnom poglavlju. Ona će nam biti temelj za sve buduće akcije.

U narednim primjerima bit će pokazan praktičan rad ADO.NET-a sa SQL Serverom te će stoga biti korištene komponente *SqlConnection*, *SqlDataAdapter* i *SqlCommand*, no imajte na umu da ćete na isti način raditi i s ostalim komponentama, ako se odlučite za rad ODBC, OLE DB ili Oracle bazama podataka.

Stvaranje veze s bazom

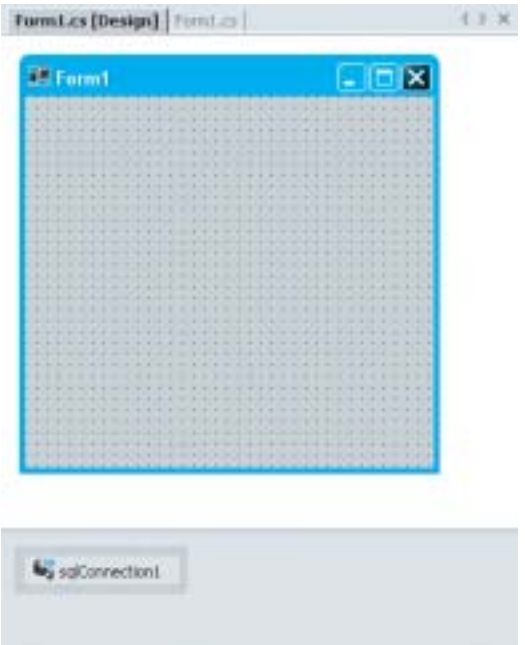
Da biste radili s bazom podataka, trebat ćemo stvoriti vezu na nju. U lijevom dijelu sučelja Visual Studija .NET nalazi se *Toolbox* prozor i u njemu *Data* okno, prikazano na slici 9-1.



Slika 9-1:
Data dio Toolbox prozora sadrži komponente za rad s bazama podataka.

Iz *Data* dijela povucite na radnu površinu aplikacije *SqlConnection* objekt, čime ga dodajete u aplikaciju. Kako se ne radi o kontroli, već o komponenti, ona će biti prikazana u posebnoj površini ispod aplikacije, kao što je prikazano na slici 9-2, i bit će joj dodijeljeno početno ime *sqlConnection1*. Jednostavnosti radi, svim ćemo u aplikaciju dodanim komponentama ostaviti početna imena.

Sljedeći korak je namještanje postavki veze s bazom. Označite dodanu komponentu *sqlConnection1* i prebacite se u *Properties* prozor. U njemu ćete pronaći *ConnectionString* svojstvo – kliknite na njega i u padajućem izborniku odaberite opciju *New Connection*. Otvorit će vam se prozor poput onog prikazanog na slici 9-3.

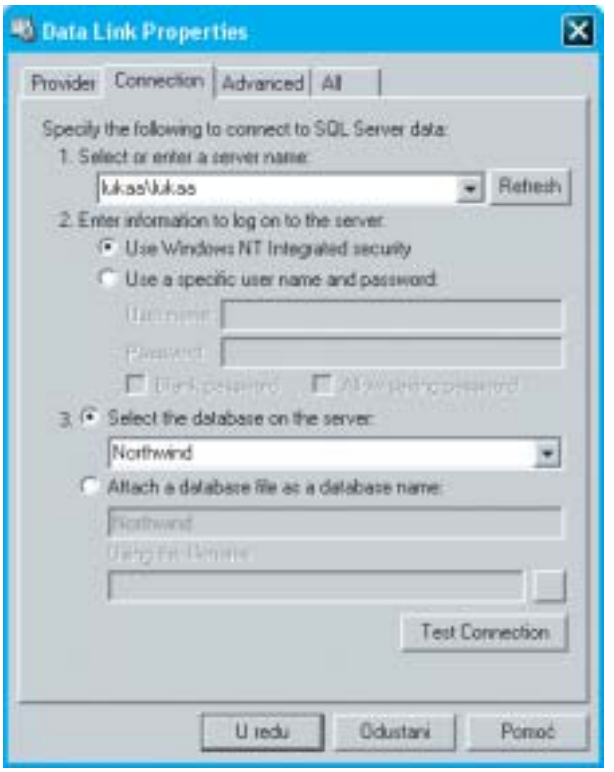


Slika 9-2:
SqlConnection kontrola pojavit će se ispod radne površine aplikacije.

Slika 9-3:
Prozor "Data Link Properties" služi za namještanje postavki veze s bazom podataka.

Želite li vidjeti na koje se sve tipove baza podataka možete spajati odnosno koji su vam sve *provideri* dostupni, kliknite na karticu *Provider* prozora "Data Link Properties" (na slici 9-3).

U našem primjeru spajat ćemo se na lokalni SQL Server i njegovu Northwind bazu podataka. Stoga u prvom koraku upišite ili odaberite ime servera iz padajućeg izbornika. U drugom koraku možete upisati korisničko ime i zaporku za spajanje na bazu ili odabrati korištenje *Integrated Securityja*. Treći korak služi za odabir baze na koju ćete se spajati.



Slika 9-4:
Postavljene sve opcije za spajanje na bazu podataka

Nakon što ste postavili sve opcije, obavezno provjerite njihovu ispravnost – kliknite na gumb *Test Connection.* Ukoliko vam se pojavi poruka "Test connection succeeded", sve je u redu i možete nastaviti s programiranjem.

Jednom načinjenu vezu s bazom podataka možete vrlo lako isprobati u *Server Explorer* prozoru koji se nalazi odmah uz *Toolbox* prozor s komponentama za aplikaciju. U njemu možete vidjeti strukturu baze, pogledati koje spremljene procedure i tablice sadržava, saznati njihova polja, tj. stupce, te dohvatiti njihov sadržaj – samo dvaput kliknite na ime tablice u *Server Exploreru*. Iskušajte sami i uvjerit ćete se da vam razvojna okolina Visual Studija pruža sve potrebno za rad, a više ne trebate u drugim programima petljati po bazi i informirati se o njihovu sadržaju ili proučavati koje stupce sadrže da ih uključite u svoje SQL upite. Ukoliko pak radite sa SQL Serverom, iz *Server Explorera* možete stvarati nove spremljene procedure, poglede ili funkcije – samo kliknite desnim gumbom miša i odaberite odgovarajuću opciju.

Zatvorite prozor za namještanje postavki veze klikom na OK. U *ConnectionString* svojstvu u *Properties* prozoru moći ćete pronaći nešto poput sljedeće generirane postavke za spajanje na bazu podataka:

```
workstation id=IME_RACUNALA;packet size=4096;integrated security=SSPI;data
source="IME_POSLUZITELJA";persist security info=False;initial catalog=Northwind
```

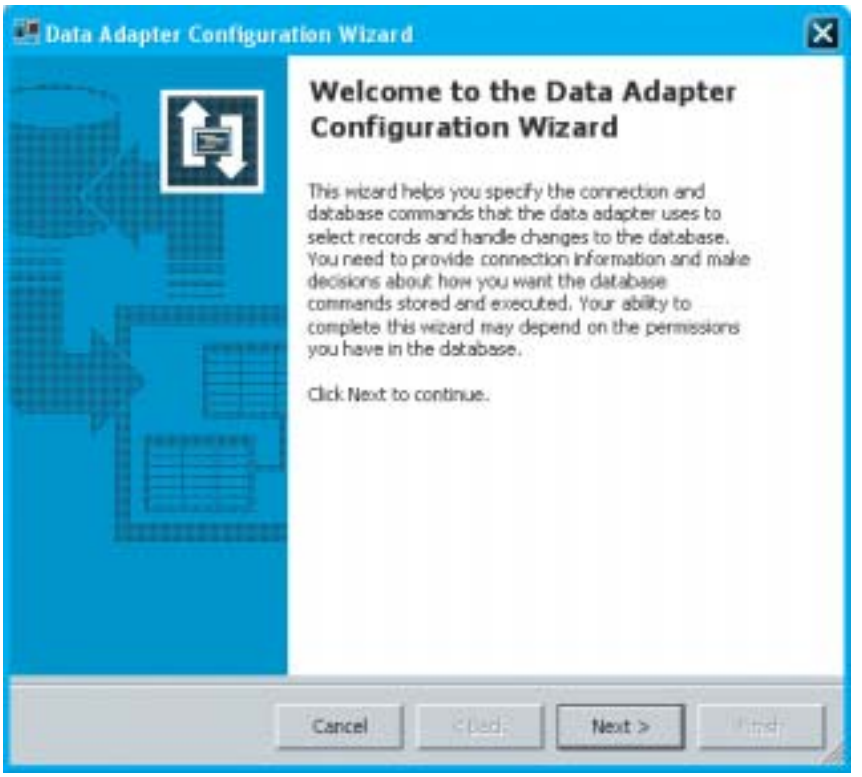
Time ste stvorili vezu s bazom podataka. U sljedećim koracima možete nastaviti programiranje i prikazivanje podataka iz baze u aplikaciji.

Upravo smo iskoristili razvojnu okolinu Visual Studija .NET za obavljanje jednostavnih akcija poput stvaranja veze s bazom, a on je u pozadini sam generirao programski kôd. Stoga je preporučljivo nakon svake ovdje objašnjene akcije prebaciti se u kôd aplikacije te potražiti i proučiti generirani kôd koji se u ovom slučaju skriva unutar regije "Windows Form Designer generated code".

Prikaz podataka u *DataGrid* kontroli

Da biste iskoristili načinjenu vezu s bazom za dohvat podataka, trebat će nam jedna komponenta *SqlDataAdapter*. Baš kao što ste učinili i s komponentom *SqlConnection* povucite i nju na radnu površinu, što će stvoriti novi objekt *sqlDataAdapter1*.

No pritom će vas dočekati čarobnjak koji će vam pomoći u dohvaćanju podataka. Otvorit će se prozor, prikazan na slici 9-5, koji predstavlja prvi korak definiranja *DataAdaptera*.



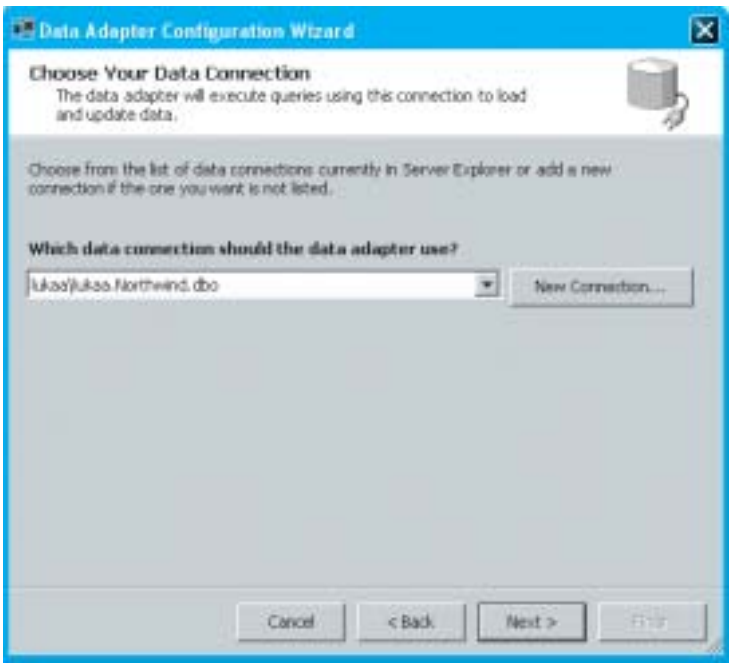
Slika 9-5: ***Prvi korak čarobnjaka za definiranje DataAdaptera***

Kliknite na *Next* i doći ćete na sljedeći korak stvaranja *DataAdaptera*. U njemu odabirete vezu na bazu podataka. Imate mogućnost stvaranja nove veze klikom na *New Connection* gumb ili odabira postojeće iz padajućeg izbornika. Kako smo mi netom stvorili novu vezu, iskoristit ćemo nju – odaberite je iz padajućeg izbornika kao na slici 9-6.

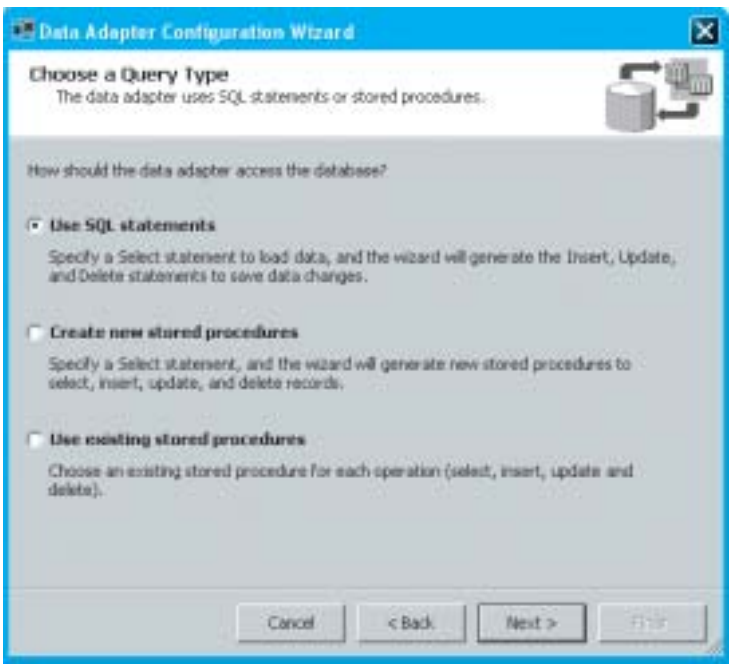
Sljedeći korak predstavlja izbor komunikacije s bazom podataka – on se može vršiti preko običnih SQL naredbi, preko spremljenih procedura koje ćete tek stvoriti ili onih već stvorenih. Jednostavnosti radi, napisat ćemo svoju SQL naredbu za dohvat podataka, a vi možete eksperimentirati i s drugim izborima.

U tekstualno polje upišite SQL upit pomoću kojeg želite dohvaćati podatke iz baze:

```
SELECT ProductID, ProductName, UnitPrice FROM Products
```



Slika 9-6: ***Odabir veze s bazom podataka***



Slika 9-7: ***Odabir načina komunikacije s bazom podataka***

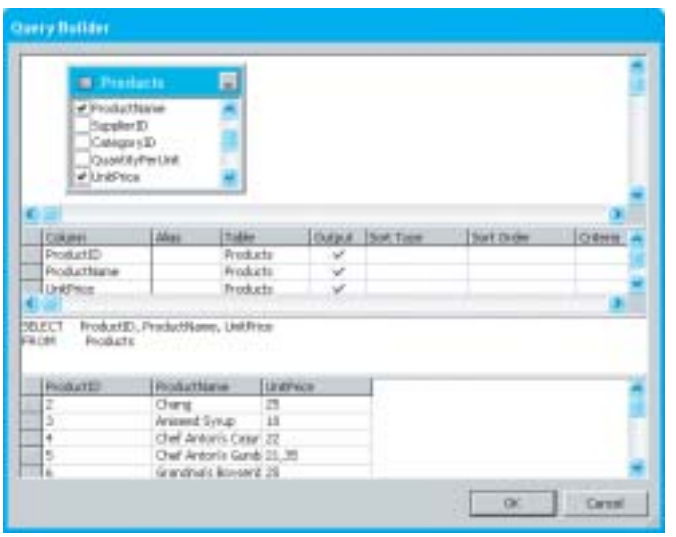
Zidam kuću trokatnicu...

U prozoru prikazanom na slici 9-9 možete kliknuti na Query Builder i pojavit će vam se prozor za složeniju izradu upita. Takav prozor je standardan za Visual Studio .NET pa do njega možete doći iz različitih dijelova aplikacije i pri obavljanju različitih poslova s bazom podataka. Isti takav prozor možete koristiti i u Microsoft Accessu za stvaranje upita ili čak Reporting Services alatu za izradu izvještaja.

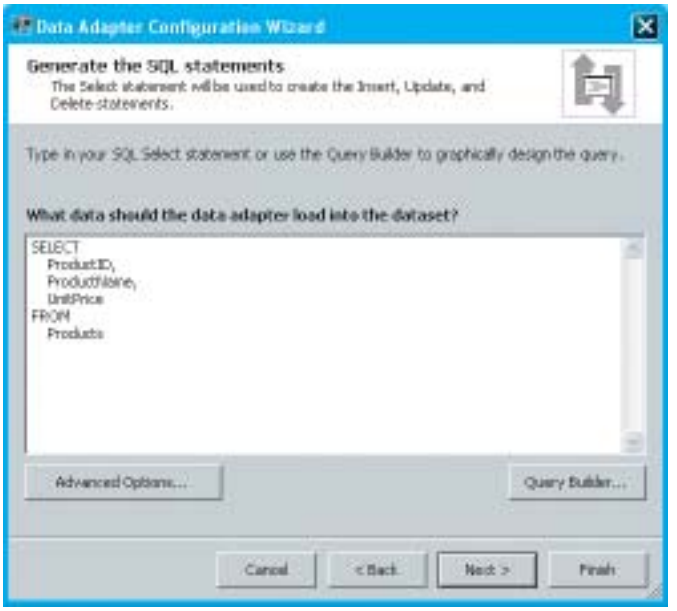
No nije to sve toliko važno – najvažnije je da vam on može uvelike pomoći pri izradi upita. Podijeljen je na 4 dijela. U prvom gornjem dijelu nalaze se tablice koje su uključene u vaš upit. Kliknete li taj dio desnim gumbom miša, možete odabrati opciju *Add Table* i dobit ćete popis svih tablica u bazi koje možete iskoristiti pri izgradnji upita. Nakon što dodate koju tablicu, polja koja želite uključiti medu dohvaćenim podacima jednostavno uključite klikom na *checkbox* u prozorčiću s tablicom.

U drugom dijelu Query Buildera možete postaviti dodatne kriterije nad stupcima. No, ukoliko ste vični ručnom pisanju SQL upita, najkorisniji će vam biti treći dio u kojem možete direktno upisati naredbu.

Kliknete li bilo gdje unutar Query Buildera desnim gumbom miša možete odabrati opciju *Run*, što će rezultirati izvršavanjem vašeg upita nad bazom podataka i dohvaćanjem podataka u četvrti dio Query Buildera. Tako možete odmah provjeriti svoj upit. Kad završite s izradom SQL upita u Query Builderu kliknite na OK i vratit ćete se u prozor na slici 9-9.

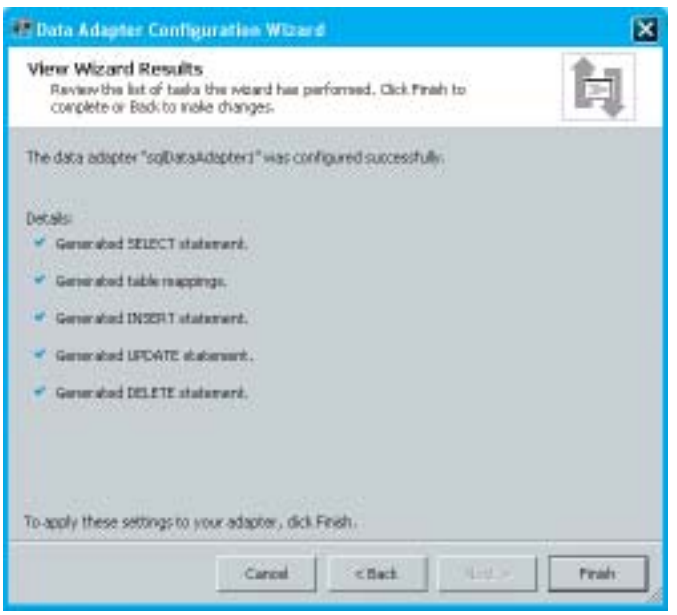


Slika 9-8: ***Query Builder može uvelike pomoći pri izgradnji SQL upita***



Slika 9-9: ***SQL upit izrađen ručno ili uz pomoć Query Buildera***

Nakon što ste izgradili upit u prozoru na slici 9-9, možete kliknuti na *Next*. Time dolazite do zadnjeg koraka i generiranja kompletnog kôda automatski. Za vas će se, ako je sve u redu, stvoriti komande za izvršavanje SELECT, INSERT, UPDATE i DELETE naredbi nad podacima u bazi. Kasnije



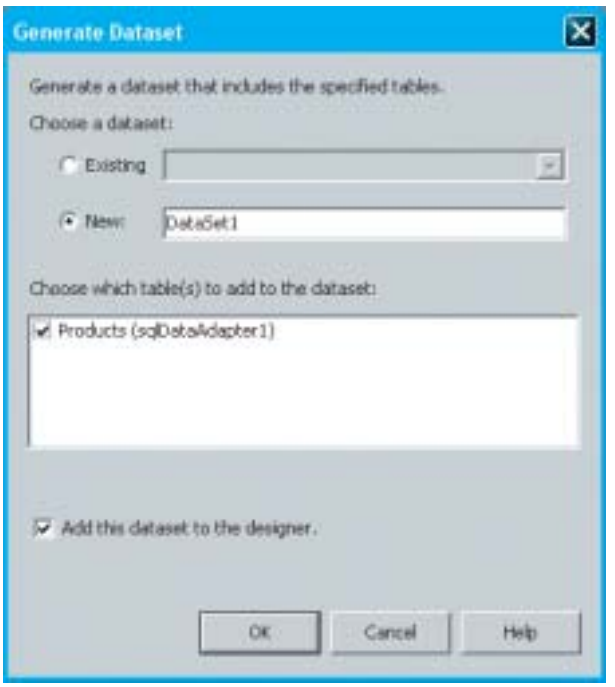
Slika 9-10: ***Zadnji korak definiranja DataAdaptera***

ćete vidjeti kako te naredbe iskoristiti – čak nećete ni morati znati da one u pozadini postoje, jer će se automatski pozivati izmjenom podataka u aplikaciji. Kliknite na *Finish*.

Nakon definiranja *DataAdaptera*, uvijek je korisno provjeriti ispravnost njegova rada. U Visual Studiju kliknite na opciju *Data – Preview Data*. Otvorit će vam se prozor u kojem možete prikazati podatke tako da kliknete na gumb *Fill Dataset* i doista se uvjerite radi li sve ispravno.

Da biste dohvaćene podatke iz baze podataka prezentirali u svojoj aplikaciji, trebat ćete ih uobličiti u *DataSet* objekt. Naravno, ni to nećete morati raditi sami, već ćete prepustiti Visual Studiju da za vas generira *DataSet* objekt.

Odaberite opciju *Data – Generate DataSet* i otvorit će vam se prozor kao na slici 9-11. U njemu morate odabrati koje ćete podatke dodati u novi *DataSet*. Dosad smo stvorili samo jedan *DataAdapter* pa ćemo njegove podatke prebaciti u *DataSet*. Kao što ste mogli vidjeti ranije, radi se o tablici *Products* iz baze *Northwind*.



Slika 9-11:
Automatsko generiranje objekta DataSet iz Visual Studija

Klikom na OK, među komponentama aplikacije ćete sad imati objekte *sqlConnection1*, *sqlDataAdapter1* i *dataSet11*.

Pri generiranju objekta *DataSet*, među datotekama u vašem projektu stvorit će se i istoimena datoteka s ekstenzijom XSD (u našem slučaju, "DataSet1.xsd") koja sadržava shemu podataka iz *DataSeta*.

Sad možemo podatke dohvaćene u objekt *DataSet* prikazati u aplikaciji. Za to će nam poslužiti *DataGrid* komponenta. Nju možete pronaći u dijelu *Windows Forms* prozora *Toolbox*. Povucite je na radnu površinu i po volji promijenite veličinu.

Označite zatim netom dodanu komponentu *DataGrid* i prebacite se u prozor *Properties*. U njemu pronađite svojstvo *DataSource* (nalazi se pod *Data*) i iz padajuće liste odaberite podatke koje želite da taj *DataGrid* prikazuje. Ako ste sve radili kao i mi, u izborniku će vam biti ponuđen *dataSet11.Products*.

Nakon što odaberete *DataSource*, stupci *DataGrida* automatski će se promijeniti i poprimiti vrijednosti prema shemi podataka iz *DataSeta*. No pokušate li sada pokrenuti aplikaciju, vidjet ćete da se podaci ipak još ne pokazuju u *DataGridu*. To je i logično – dosad ste tek povezali *DataGrid* s izvorom podataka, no te podatke još niste dohvatili.

Kliknite dvaput na vašu aplikaciju ili se prebacite unutar metode *Form_Load* u kodu. Tamo je potrebno upisati sljedeći kôd:

```
sqlDataAdapter1.Fill(dataSet11);
```

Dakle, *dataSet11* s kojim je i povezan *DataGrid* u aplikaciji je inicijalno prazan. Po učitavanju aplikacije trebamo ga napuniti, a to činimo metodom *Fill* našeg *DataAdaptera* u kojem smo definirali kako dohvatiti podatke. Pokrenete li sada aplikaciju, dobit ćete nešto kao na slici 9-12.

Slika 9-12:
***Tablica prikazana u* DataGridu**

Tablica prikazana u *DataGridu* može imati i svoj naslov umjesto prazne trake prikazane na slici 9-12. Među svojstvima *DataGrida* pronađite svojstvo *CaptionText* i postavite ga na neku vrijednost, primjerice "Proizvodi".

No podatke dohvaćene ovakvim *DataSetom* možete i uređivati. Još jednom označite *DataGrid* i provjerite je li mu svojstvo *ReadOnly* u *Properties* izborniku postavljeno na *False*. Ako nije, postavite ga, jer želimo *DataGrid* iskoristiti za uređivanje podataka u bazi, što uključuje dodavanje novih podataka i brisanje postojećih.

Dodajte u aplikaciju pored *DataGrida* i jedan gumb (ostavite mu *defaultno* ime "button1"). Promijenite mu tekst u "Spremi promjene", jer će on, kao što mu i ime kaže, služiti za spremanje promjena načinjenih u *DataGridu*. Kliknite na njega dva puta i upišite naredni kôd u metodu *button1_Click*:

```
int num = sqlDataAdapter1.Update(dataSet11);
MessageBox.Show("Promijenjeno je " + num.ToString() + " zapisa.", "Operacija
  završena");
```

Pozivamo metodu *Update* koja koristi načinjeni *DataAdapter* za izmjenu svih podataka u *dataSet11* objektu. Prisjetimo se, taj je objekt povezan s *DataGridom* i u njemu se nalaze sve promjene koje je korisnik načinio nad podacima u aplikaciji.

Metoda *Update* vraća cijeli broj koji je jednak broju promijenjenih redaka. Njega spremamo u varijablu *num* i koristimo pri ispisu poruke o broju promijenjenih zapisa.

No pri izmjeni podataka u bazi može ponekad doći i do greške – primjerice, možete pokušati izbrisati podatak čiji se ID referencira u nekoj drugoj tablici, čime biste narušili integritet podataka. To baza neće dopustiti, a vama će se u aplikaciji javiti greška.

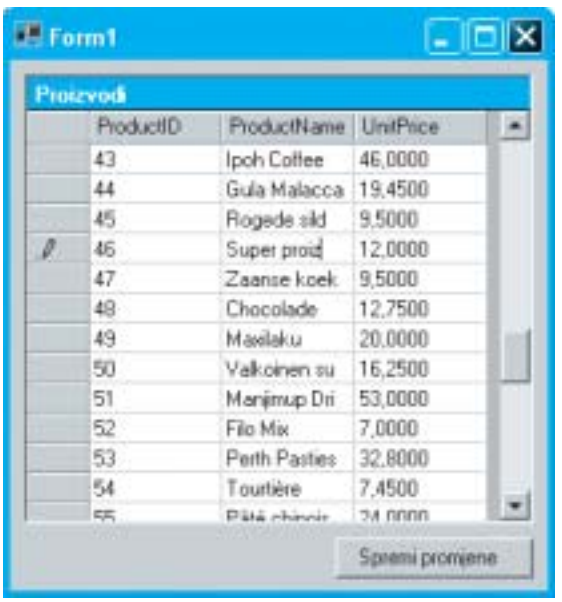
Stoga je prikladno napisati kôd koji će provjeravati greške i shodno tome se ponašati. No bit ćemo jednostavni – u slučaju greške ispisat ćemo poruku koju nam vrati baza podataka, a ona će sadržavati razlog neuspjeha metode *Update*.

```
int num;
try
{
  num = sqlDataAdapter1.Update(dataSet11);
}
catch (SqlException se)
{
  MessageBox.Show(se.Message, "Greška!");
  num = 0;
}
```

```
MessageBox.Show("Promijenjeno je " + num.ToString() + " zapisa.", "Operacija
  završena");
```

Dakle, u bloku *try* pokušavamo izvršiti metodu *Update*, a u bloku *catch* hvatamo iznimku *SqlException* (svakako provjerite imate li na početku programa "using System.Data.SqlClient"). Ukoliko se ona pojavi, ispisujemo njen tekst, jer se radi o grešci. Na samom kraju ispisujemo broj izmijenjenih podataka.

Kako se uopće mogu mijenjati podaci *DataGridom*? Jednostavno. Pozicionirajte se u bilo koji redak, kliknite mišem i on će se pretvoriti u tekstualno polje čiji ćete sadržaj moći mijenjati. Tako možete mijenjati sadržaj bilo kojeg retka, no ne možete mu promijeniti primarni ključ, u našem slučaju *ProductID*.



Slika 9-13:
Naša gotova aplikacija u procesu mijenjanja sadržaja nekog retka

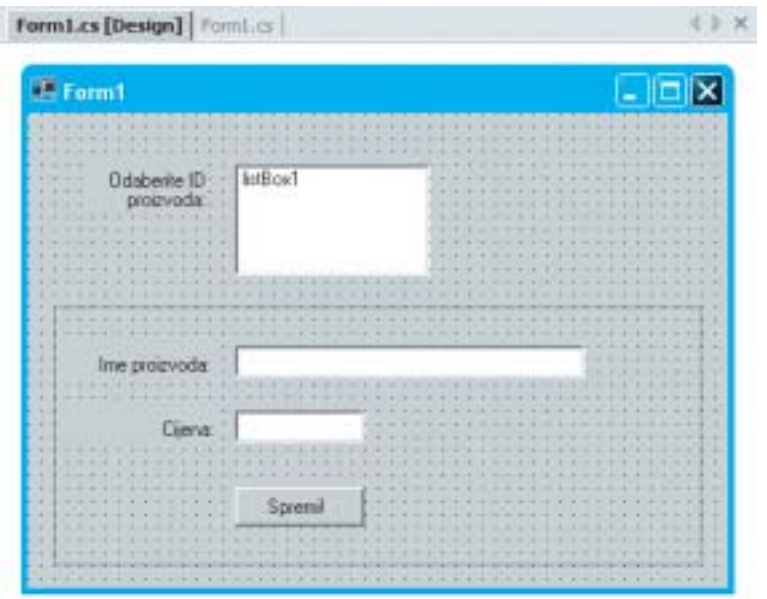
Retke možete i brisati. Označite cijeli redak i pritisnite DEL gumb na tastaturi. Želite li pak dodati novi redak, pozicionirajte se na dno *DataGrida* i uočit ćete prazan redak označen s "*". U njega možete upisati nove podatke. Kad jednom pozovete metodu *Update*, ti će se podaci automatski sinkronizirati s bazom podataka odnosno snimit će se sve promjene.

***DataGridu* možete mijenjati standardni izgled. Označite ga i na dnu prozora *Properties*, ispod tablice sa svojstvima, pronađite link *Auto Format*. Kliknete li na njega, otvorit će vam se prozor s mogućim temama izgleda *DataGrida*. Pronađite onu koja vam se najviše sviđa, kliknite na OK i vaš će *DataGrid* odmah promijeniti izgled.**

Uređivanje podataka u bazi pomoću formulara

Naš sljedeći zadatak bit će uređivanje podataka prikazanih iz prošlog primjera pomoću formulara. Nećemo naglasak staviti na samu aplikaciju, već na pristup podacima – cilj nam je pokazati praktičnu uporabu *DataReadera* te objekata *Command* s parametrima i bez njih.

Izradite stoga formular izgleda poput onog na slici 9-14. U gornji dio stavite kontrolu *ListBox* u kojoj će stajati popis zapisa iz baze. Odabirom nekog zapisa iz baze, njegovo ime i cijena će se učitati u dva tekstualna polja spremljena niže u formularu. Klikom na gumb *Spremi*, novi će se podaci spremiti u bazu podataka. Svim kontrolama smo ostavili *defaultna* imena, pa se *ListBox* kontrola i dalje zove *listBox1*, tekstualno polje za ime proizvoda *textBox1*, za cijenu *textBox2,* a gumb za spremanje *button1*.



Slika 9-14: ***Kontrole u formularu za uređivanje podataka iz baze***

Dohvaćanje popisa zapisa

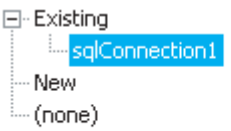
Krenut ćemo standardnim načinom. Na radnu površinu povucite objekt *SqlConnection* iz dijela *Data* prozora *Toolbox*. Izgradite novi *connection string* ili odaberite neki već postojeći iz padajućeg izbornika u svojstvu *ConnectionString* u prozoru *Properties*. Kao što smo rekli, dohvaćat ćemo i mijenjati podatke kao u prošlom primjeru, dakle iz tablice *Products* iz baze *Northwind*.

U svojoj aplikaciji možete imati više veza na različite baze podataka, tj. *SqlConnection* objekata. Tako smo podatke u našem formularu mogli dohvaćati iz više različitih baza ili izvora podataka, samo nam trebaju dodatni *Connection* objekti, a postupak i dalje ostaje isti.

No, za razliku od prethodnog primjera u kojem smo koristili *DataAdapter*, ovaj put ćemo iskoristiti objekte *Command* za komunikaciju s bazom. Ukupno će nam trebati čak tri objekta *Command* – jedan će dohvaćati popis svih zapisa u bazi koji će se puniti u izbornik u gornjem dijelu formulara, drugi će dohvaćati podatke nekog zapisa (ime proizvoda i cijenu) u trenutku kad korisnik odabere drugi proizvod u izborniku, a treći će vršiti naredbu UPDATE i mijenjati podatke u bazi.

Stoga napravimo prvi objekt *Command* – povucite iz *Data* dijela prozora *Toolbox* na radnu površinu objekt *SqlCommand*. Dodat će se na dno radne površine među ostale objekte za rad s podacima.

Da bi objekt *Command* ispravno radio, treba mu namjestiti vezu s bazom podataka, tj. odabrati koja se veza koristi. Kako smo mi već napravili vezu s bazom podataka, iskoristit ćemo postojeću. U svojstvu *Connection* u padajućem izborniku odaberite *Existing – sqlConnection1*, kao što je i prikazano na slici 9-15.



Slika 9-15:
Odabir veze s bazom podataka za objekt Command

Objekt *Command* vezan uz SQL Server može pozivati spremljene procedure, a može i izvršavati obične SQL upite. Zbog jednostavnosti, u primjerima ćemo sami izgrađivati SQL upite. Stoga provjerite je li u svojstvu *CommandType* odabran "Text", a zatim u svojstvu *CommandText* upišite novi SQL upit ili ga načinite pomoću već opisanog Query Buildera.

Kako namještamo prvi objekt *Command* koji će služiti za dohvaćanje podataka u *ListBox*, napisat ćemo sljedeći upit:

```
SELECT ProductID FROM Products ORDER BY ProductID
```

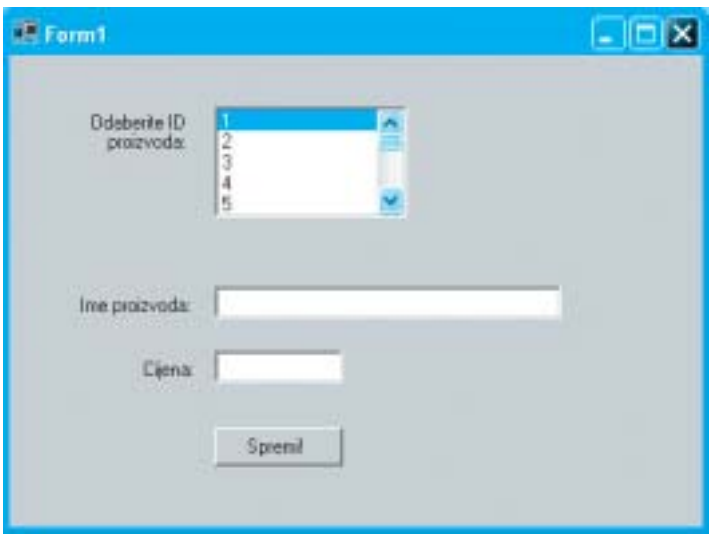
Još nam je samo ostalo zbog jednostavnosti promijeniti ime objekta *Command*. Promijenite mu stoga svojstvo *(Name)* i u njega upišite *cmdProizvodi*. Podsjetimo, imat ćemo tri objekta *Command,* pa ih je nužno tako nazvati kako bismo se lakše snalazili.

Sljedeći korak je pozvati načinjeni objekt *Command* i upisati dohvaćene podatke u kontrolu *ListBox*. To ćemo obaviti odmah po učitavanju formulara – kliknite dvaput bilo gdje na površinu formulara i prebacit ćete se u metodu *Form_Load* koja se izvršava po učitavanju, dakle točno ono što nama treba.

U nju upišite sljedeći kôd:

```
sqlConnection1.Open();
SqlDataReader dr = cmdProducts.ExecuteReader();

while (dr.Read())
{
 listBox1.Items.Add(dr["ProductID"].ToString());
}

dr.Close();
sqlConnection1.Close();

listBox1.SelectedIndex = 0;
```

Krenimo redom. Na početku otvaramo vezu s bazom jer nam je ona potrebna za dohvaćanje podataka, a zatim stvaramo objekt tipa *SqlDataReader* i u njega spremamo rezultate izvršavanje načinjenog objekta *Command*. I tu je ključ – pozivom metode *ExecuteReader()* nad objektom *Command* dohvaćamo podatke u *DataReader* i njih možemo čitati i koristiti u aplikaciji.



Slika 9-16: ***Prva verzija aplikacije koja podacima iz baze popunjava kontrolu ListBox***

U petlji *while* prolazimo kroz sve dohvaćene zapise – naredbom *Read()* nad *DataReaderom* čitamo naredni zapis i sve dok je ona *true*, zapisi postoje. U trenutku kad ona vrati *false*, došli smo do kraja učitanih zapisa i vrijeme je da petlja prestane s izvršavanjem.

U samoj petlji dodajemo nove elemente u kontrolu *ListBox* pozivanjem metode *Add*. Kao elemente dodajemo vrijednosti *ProductID* svakog zapisa u bazi. Njih dohvaćamo već poznatim načinom – korištenjem *DataReadera* i navođenjem imena stupca unutar navodnika u uglatim zagradama. Sve naravno pretvaramo u znakovni niz da bi se moglo prikazati u aplikaciji.

Nakon petlje zatvaramo otvoreni objekt *DataReader* i vezu s bazom podataka. Također, označavamo u kontroli *ListBox* prvi zapis, što će nam trebati kasnije. Indeksi svih zapisa, kao i u drugim kolekcijama, kreću od nule, pa prvi zapis ima indeks 0.

Pokušajte pokrenuti aplikaciju i uvjerit ćete se sami – rezultat je prikazan na slici 9-16.

Dohvaćanje informacija o jednom zapisu

Krenimo s nadogradnjom naše jednostavne aplikacije. U sljedećem koraku ćemo pokazati kako se pozivaju objekti *Command* s parametrima jer želimo dohvatiti informacije o jednom zapisu iz baze, ovisno o tome koji se proizvod odabere u *ListBoxu*.

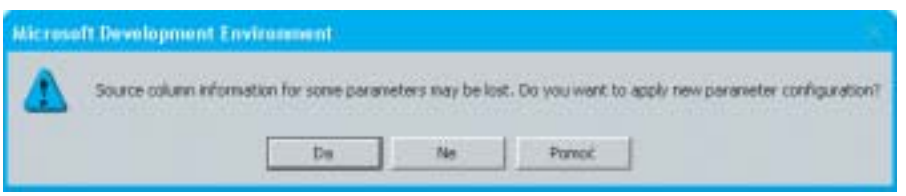
Stoga dodajte još jedan objekt *Command* u aplikaciju, a ime mu promijenite u *cmdGetProduct* (primijetite da svuda za objekte *Command* koristimo prefiks "cmd" kako bismo se lakše snalazili među raznim objektima koje koristimo). Kao i u prethodnom primjeru, postavite mu *Connection* svojstvo na već postojeću vezu na bazu podataka, *sqlConnection1*.

Ključan je upis SQL upita. Otvorite Query Builder iz svojstva *CommandText* i u njemu upišite sljedeći upit:

```
SELECT ProductName, UnitPrice FROM Products WHERE (ProductID = @PID)
```

Kao što vidite, dohvaćamo ime i cijenu za onaj proizvod kojem vrijednost *ProductID* odgovara vrijednosti @PID, što predstavlja parametar imena "PID" (nazvali smo ga tako, jer sadržava vrijednost

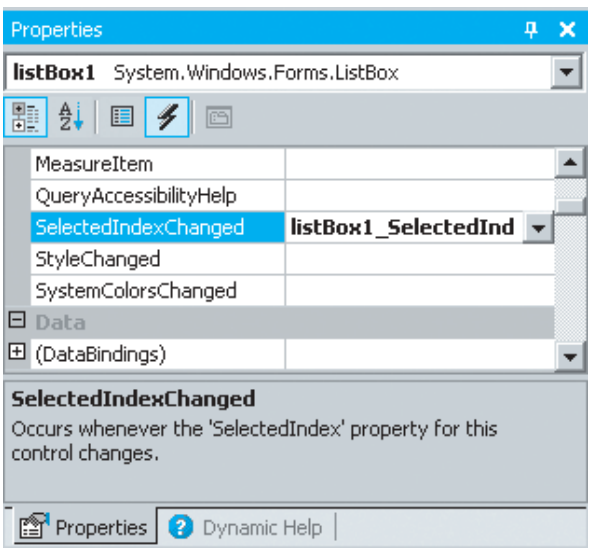
Slika 9-17:
Automatsko stvaranje kolekcije parametara za objekt Command



ProductID-a). Parametri se, dakle, označavaju znakom "@". U našoj aplikaciji ćemo u različitim situacijama davati drugu vrijednost parametru @PID, što će rezultirati drugačijim SQL upitom i, samim time, drugačijim rezultatima.

Svaki objekt *Command* ima svoju kolekciju parametara. U prethodnom primjeru ih nismo koristili, no sada to činimo. Nakon što upišete SQL upit koji koristi parametre, Visual Studio će za vas automatski generirati kolekciju parametara. Prikazat će vam se prozor na slici 9-17, a vi potvrdno odgovorite jer želite koristiti parametre definirane u SQL upitu.

Sad trebamo na promjenu odabranog proizvoda u *ListBoxu* promijeniti i sadržaj tekstualnih polja. Za to ćemo se pouzdati u događaj u aplikaciji odnosno nad samim *ListBoxom*. Označite ga te se prebacite u prozor *Properties*. U njemu, u dijelu *Events*, kao što je prikazano na slici 9-18, dvaput kliknite u svojstvo *SelectedIndexChanged*. Automatski će se stvoriti nova metoda koja će se pozivati na svaku promjenu odabranog elementa u *ListBoxu*.



Slika 9-18: ***Događaj SelectedIndexChanged poziva se pri svakoj promjeni odabranog elementa u ListBoxu.***

U novoj metodi koja se, dakle, poziva na svaku promjenu odabranog elementa u *ListBoxu* trebat ćemo dohvatiti informacije iz baze za proizvod s odabranim ID-em. Evo i kompletnog kôda:

```
private void listBox1_SelectedIndexChanged(object sender, System.EventArgs e)
{
```

```
string pid = listBox1.SelectedItem.ToString();

sqlConnection1.Open();
cmdGetProizvod.Parameters["@PID"].Value = pid;
SqlDataReader dr = cmdGetProizvod.ExecuteReader(CommandBehavior.SingleRow);

if (dr.Read())
{
 textBox1.Text = dr["ProductName"].ToString();
 textBox2.Text = dr["UnitPrice"].ToString();
}

dr.Close();
sqlConnection1.Close();
}
```

Na samom početku u varijablu *pid* učitavamo vrijednost trenutno odabranog elementa u *ListBoxu*. To je ID proizvoda čije ime i cijenu želimo učitati u tekstualna polja.

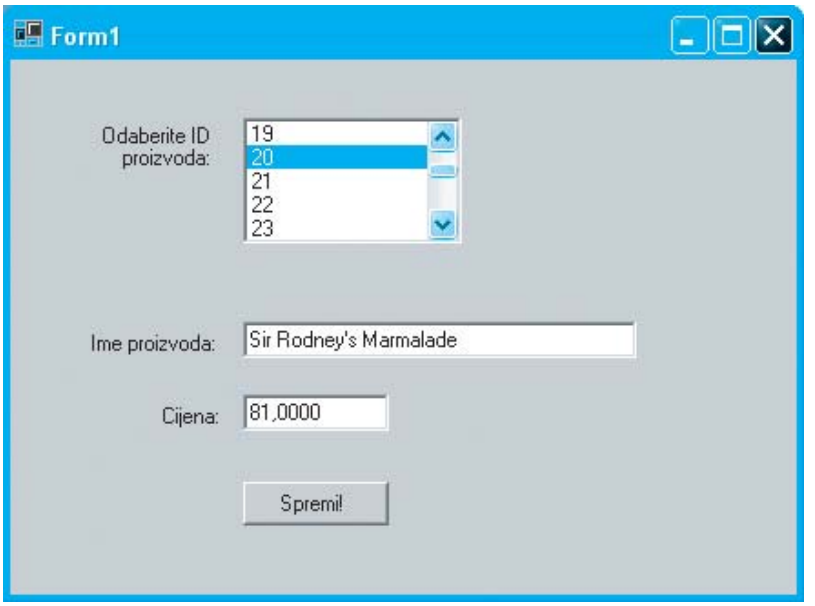
Zatim otvaramo vezu s bazom podataka. I slijedi ključni dio – objektu *cmdGetProizvod* postavljamo vrijednost parametra @PID na vrijednost trenutno odabranog elementa iz *ListBoxa*. To činimo pomoću kolekcije *Parameters,* navodeći ime parametra unutar navodnika u uglatim zagradama (nemojte zaboraviti znak "@" u imenu parametra!).

Sad tu naredbu možemo izvršiti. Kao i u prethodnom primjeru, izvršavamo naredbu *ExecuteReader* koja vraća podatke iz baze. No ovaj put naredbi prosljeđujemo jedan parametar. Radi se o pobrojanom polju imena *CommandBehavior* i njegovoj vrijednosti *SingleRow*. Naime, kako planiramo dohvatiti samo jedan zapis iz baze, navodeći pri izvršavanju naredbe *CommandBehavior.SingleRow* optimiziramo njeno izvršavanje i time poboljšavamo rad aplikacije. U prethodnom primjeru smo očekivali više vraćenih redaka, pa nismo navodili ovaj parametar.

Opet koristimo metodu *Read()* za čitanje – ukoliko ona vrati vrijednost *true*, znači da postoje podaci za čitanje, pa ih možemo upisati u tekstualna polja. U slučaju da je vratila *false*, to bi značilo da naš upit nije vratio niti jedan rezultat, što je očita posljedica neke greške, pa ne bi ni imalo smisla pokušavati pročitati podatke i zapisati ih u tekstualna polja.

Sjetite se kako smo na kraju metode *Form_Load* označili prvi element u *ListBoxu*. Time se automatski poziva maloprije napisana metoda *listBox1_SelectedIndexChanged* pa imamo sljedeći rezultat – po pokretanju aplikacije označava se prvi element, a samim time se i učitavaju vrijednosti tog zapisa iz baze i u odgovarajuća polja u formularu upisuju ime i cijena proizvoda.

Nakon zapisivanja podataka u formular, zatvaramo *DataReader* i, naravno, vezu s bazom podataka. Iskušajte sada aplikaciju – pokrenite je i mijenjajte odabrane elemente i vidjet ćete kako se imena i cijene proizvoda upisuju u polja formulara.



Slika 9-19: ***Poboljšana verzija aplikacije. S promjenom odabira u ListBoxu mijenjaju se vrijednosti u odgovarajućim tekstualnim poljima.***

Spremanje promjena

Ostalo nam je još samo spremanje promjena. Za to će nam trebati treći objekt *Command*. Promijenite mu ime u "cmdSaveProizvod" i upišite naredni SQL upit:

```
UPDATE Products SET ProductName = @PN, UnitPrice = @UP WHERE (ProductID = @PID)
```

Dakle, izvršit ćemo naredbu UPDATE koja će spremiti novo ime proizvoda (parametar @PN) i novu cijenu proizvoda (parametar @UP) za određeni zapis. Koji točno zapis treba promijeniti odredit ćemo pogledavši koji je element odabran u *ListBoxu*.

Kliknite dvaput na gumb na kontroli koji je predviđen za spremanje podataka u bazu i prebacit ćete se u metodu *button1_Click* koja će se pozvati na svaki klik na gumb.

```
private void button1_Click(object sender, System.EventArgs e)
{
 cmdSaveProizvod.Parameters["@PN"].Value = textBox1.Text;
```

```
cmdSaveProizvod.Parameters["@UP"].Value = textBox2.Text;
cmdSaveProizvod.Parameters["@PID"].Value = listBox1.SelectedItem.ToString();

sqlConnection1.Open();
int num = cmdSaveProizvod.ExecuteNonQuery();
sqlConnection1.Close();

if (num == 1)
{
 MessageBox.Show("Promjene su uspješno snimljene.", "Uspjeh");
}
}
```

Najprije postavljamo parametre naredbe *cmdSaveProizvod*. Parametar @PN služi za spremanje vrijednosti *ProductName*, a nju čitamo iz prvog tekstualnog polja. S druge strane, parametar @UP služi za spremanje vrijednosti *UnitPrice*, a nju čitamo iz drugog tekstualnog polja. Treći parametar određuje ID proizvoda čije ime i cijenu trebamo promijeniti, a njega čitamo iz *ListBoxa* i on odgovara trenutačno označenom ID-u.

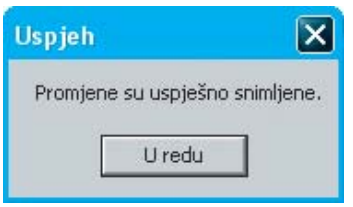
Dalje je sve poznato – otvaramo bazu podataka te nad objektom *Command* izvršavamo metodu *ExecuteNonQuery()* koja služi za izvršavanje naredbi koje ne vraćaju neke podatke (dakle, nećete je upotrijebiti uz naredbu SELECT), već vraća broj zapisa koji su promijenjeni (što može podrazumijevati dodane zapise, ukoliko izvršavate naredbu INSERT, izbrisane ukoliko izvršavate DELETE ili izmijenjene ukoliko pak izvršavate naredbu UPDATE).

Dosad smo objasnili pozivanje *ExecuteReader()* i *ExecuteNonQuery()* metoda nad *Command* objektom, no još jedna vam može biti veoma korisna. Primjerice, želite li dohvatiti samo jednu vrijednost odnosno vrijednost prvog stupca u prvom zapisu dohvaćenih podataka, korisna će vam biti *ExecuteScalar()* naredba. Primjerice, imate li SQL naredbu "SELECT COUNT(*) FROM Products" kojom brojite zapise u tablici *Products*, nju biste izvršili na sljedeći način i odmah u varijabli *broj* imali traženi podatak jer je on dohvaćen kao vrijednost prvog stupca u prvom zapisu podataka. Ne treba ni govoriti koliko je to brži način dohvaćanja jednog podatka iz baze od, primjerice, korištenja *DataReader* objekta za takve stvari.

```
int broj = cmdBrojProizvoda.ExecuteScalar();
```

Podatak o broju izmijenjenih zapisa spremamo u varijablu *num*. Ukoliko je njena vrijednost jednaka 1, znači da smo uspješno izmijenili samo jedan zapis, kao što je i planirano. Na kraju ispisujemo poruku o uspjehu.

Slika 9-20:
Uspješno obavljena operacija spremanja



Korištenje ADO.NET-a i rad s podacima iz baza podataka ide ruku pod ruku s *error handlingom* i hvatanjem iznimki. Preporučuje se svaku operaciju s bazom podataka staviti u blok *try catch* i, naravno, hvatati samo odgovarajuće iznimke, primjerice tipa *SqlException*. Greške pri radu s ADO.NET-om su moguće jer ne ovise o vašem kodu – primjerice, baza podataka može biti nedostupna, SQL Server može biti zaustavljen i ne odgovara na vaše upite, pokušavate izbrisati zapis koji ne možete izbrisati zbog referencijalnog integriteta baze i slično. U svakom slučaju, iako smo ovdje radi jednostavnosti izbjegli pisanje blokova *try catch*, to preporučujemo u stvarnim situacijama.

Na kraju, sve ovdje opisano vrijedi i za ASP.NET web-aplikacije. Iako smo u primjerima koristili prozorske aplikacije, ADO.NET je ključan dio .NET *frameworka* i možete ga iskoristiti u svim tipovima aplikacija, bilo da izrađujete ASP.NET aplikacije, web-servise ili aplikacije za mobilne uređaje.

10. POGLAVLJE

ASP.NET

U ovom poglavlju:

- Kako pisati web-aplikacije
- Tipovi kontrola i njihovo korištenje
- Što su aplikacija i sesija
- Konfiguracija servera i aplikacija
- *Cacheiranje* izlaza, fragmenata i podataka
- Povezivanje s ADO.NET-om

Vjerujemo da o značaju i korisnosti web-aplikacija ne treba trošiti riječi. Štoviše, vjerujemo da će dio čitatelja knjige preskočiti neka (po njihovu sudu) manje zanimljiva poglavlja i odmah skočiti na igranje sa zlatnom kokom – na izradu web-aplikacija. Zato prije nego što zaronimo u detalje moramo zamoliti takve čitatelje da odustanu od te namjere i čitaju poglavlja redom. Naime, jedna od glavnih promjena koje je donio ASP.NET jest značajno približavanje programskom modelu koji smo upoznali u poglavlju o izradi prozorskih aplikacija, pa bi preskakanje istog moglo biti kobno.

Sukladno s općenitim stavom u knjizi, nećemo u detalje uspoređivati ASP.NET sa starijim inačicama ove tehnologije, no valja naglasiti da su se stvari značajno promijenile. Prvenstveno

stoga što je ASP.NET objektno orijentiran i bolje strukturiran. Naravno, to ne znači da će vaše eventualno poznavanje ASP-a biti beskorisno, no morate imati na umu da će pristup pisanju web-aplikacija biti drastično izmijenjen.

Nostalgija za kôdom

Što učiniti s hrpom već postojećeg kôda i skriptama u ASP-u koje godinama uspješno rade? Prelazak na ASP.NET preporučuje se iz mnogo razloga, a na vrhu liste nalaze se performanse i stabilnost.

Iako je Microsoft pazio da "prevođenje" kôda bude što lakše, ipak se radi o prilično dugotrajnom i napornom poslu. Stoga je dobrodošla mogućnost suživota web-aplikacija pisanih u ASP-u i ASP.NET-u na istom računalu, pa čak i na istim web-stranicama. Doduše, oni neće moći dijeliti stvari poput aplikacijskih i sesijskih varijabli, niti na isti način pristupati bazama podataka, no mogućnost suživota olakšat će vam utoliko što ćete prebacivanje moći raditi po segmentima.

Evo nekoliko glavnih stvari na koje ćete morati paziti:

- Kôd i dalje može biti smješten između *tagova* <% i %>, ali u njemu ne mogu biti deklarirane varijable; također, taj će se kôd izvršavati prilikom renderiranja stranice, nakon ostalog kôda pisanog u datoteci s pozadinskim kôdom (o tome nešto kasnije).
- U ASP-u se moglo miješati jezike unutar iste stranice, dok kod ASP.NET-a cijela stranica mora biti u istom jeziku.
- Ukoliko ste u ASP-u koristili programski jezik Basic, koristili ste VBScript. U ASP.NET-u koristi se Visual Basic .NET koji se razlikuje u sintaksi.
- U ASP-u su sve varijable bile tipa *variant*, što je značilo da mogu poprimiti vrijednost bilo kojeg tipa. U ASP.NET-u, kao i u cijelom .NET Frameworku, tip varijable mora biti strogo definiran (engl. *strongly typed*).
- ASP po *defaultu* prosljeđuje parametre kao reference, a ASP.NET kao vrijednosti.
- Parametri metoda moraju biti u zagradi, čak i ako nema povratne vrijednosti.
- Obavezno je deklariranje varijabli.

Naravno, ovo je tek vrh ledene sante. Detaljnije informacije možete naći na brojnim web-stranicama ili u MSDN-u, na putanji .NET Development > .NET Framework SDK > .NET Framework > Building Applications > Creating ASP.NET Web Applications > Migrating ASP Pages to ASP.NET ili .NET Development > ASP.NET > Technical Articles > Migrating to ASP.NET: Key Considerations.

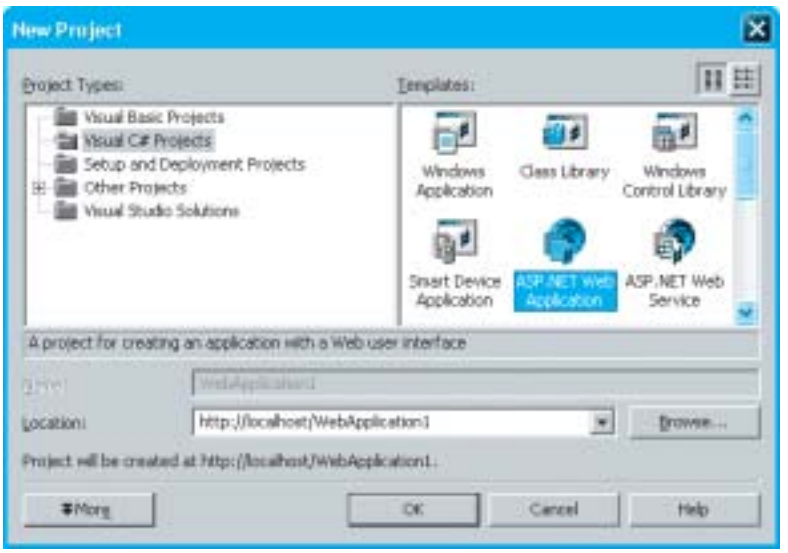
Usput, postoji i alat za, uvjetno rečeno, automatsku konverziju stranica iz ASP-a u ASP.NET. Više informacija o njemu, kao i neke druge korisne savjete, možete potražiti na adresi http://msdn.microsoft.com/asp.net/using/migrating.

Kao i kod svih aplikacija pisanih u .NET-u, i stranice ASP.NET zahtijevaju postojanje .NET Frameworka na računalu na kojem se nalaze (posjetitelji tih stranica, naravno, ne moraju imati .NET Framework). Osim njega, zbog naravi samih web-stranica, u igri će morati biti i poslužitelj web-stranica, konkretno Internet Information Server (skraćeno IIS). On je sastavni dio gotovo svih Windowsa, a za posluživanje stranica tipa ASP.NET ne treba nikakvu posebnu konfiguraciju ili podešavanje – dovoljno je da nastavak datoteka bude .aspx i stranica će biti prepoznata kao stranica tipa ASP.NET.

Nastavak .aspx potječe od prvobitnog imena ASP.NET-a koji se tada zvao ASP Plus.

Forme i kontrole

Kao kod upoznavanja s prozorskim aplikacijama, i ovdje ćemo prolaziti kroz jednostavne primjere kako bismo upoznali mogućnosti i specifičnosti izrade web-aplikacija. Prvi korak u tome je stvaranje novog projekta.



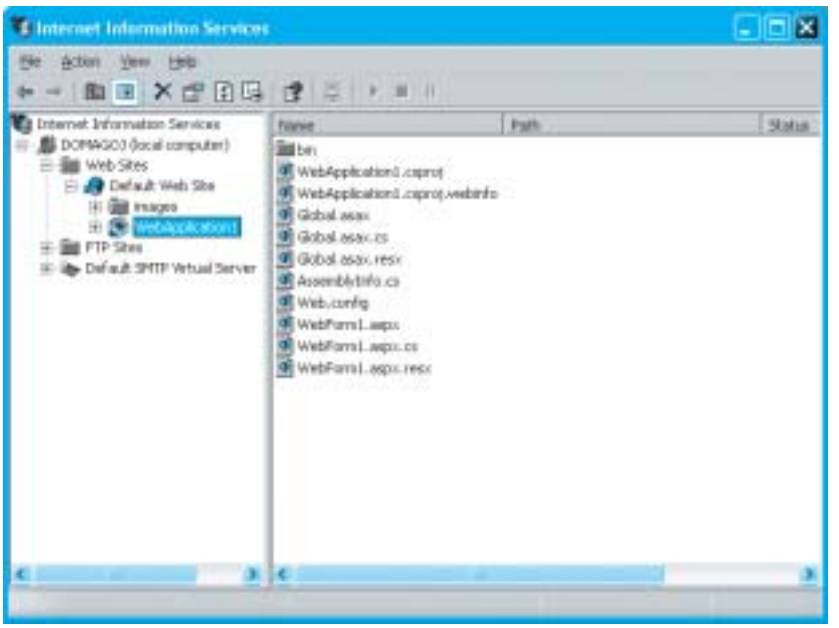
Slika 10-1: ***Stvaranje novog projekta za web-aplikaciju***

U ovom poglavlju koristit ćemo predložak nazvan ASP.NET Web Application. Nakon što ga označite (vidi sliku 10-1), primijetit ćete da u polje lokacije ne upisujemo mapu na disku, nego mapu na lokalnom web-poslužitelju.

Naime, kako smo već spomenuli, svaka aplikacija zahtijeva postojanje web-poslužitelja koji je nužan za njezin razvoj i pokretanje. Logično bi bilo da svaka aplikacija bude na zasebnoj web-adresi (poput www.bug.hr), no kako većina klijentskih inačica Windowsa ima ograničenje na samo jedan web-*site* po poslužitelju, aplikacije se smještaju u zasebne mape i dobivaju adresu poput http://localhost/ImeAplikacije/. Osim toga, smještanje na posebnu osnovnu adresu uključivalo bi konfiguraciju IP-adresa i DNS-a, pa je ovaj pristup praktičniji i jednostavniji.

Adresa http://localhost/ je adresa lokalnog računala. Ona upućuje na IP-adresu 127.0.0.1 koja predstavlja adresu lokalnog računala, koju nazivamo i *loopback*.

Nakon što kreirate novi projekt, Visual Studio će u pozadini napraviti nekoliko stvari. Prvo će kreirati mapu na disku. Ako niste dirali osnovnu konfiguraciju IIS-a, ta će se mapa nalaziti na putanji "C:\Inetpub\wwwroot\ImeAplikacije\". Na tu će mapu pokazivati već spomenuta virtualna mapa http://localhost/ImeAplikacije/ koja će biti otvorena na web-poslužitelju i predstavljat će samostalnu web-aplikaciju sa svim karakteristikama, bez obzira na adresu.



Slika 10-2: ***Novootvorena aplikacija u mapi označena je drugačijom ikonom od običnih mapa.***

Bez obzira na to što za vrijeme razvoja web-aplikaciju smještamo u mapu, ona će kasnije moći normalno raditi na poslužitelju na osnovnoj adresi. Bit će je dovoljno kopirati.

Web-forma

Slično kao i kod izrade prozorskih aplikacija, i ovdje se koriste forme, no nešto izmijenjenih karakteristika. Dok smo kod prvih formu poistovjećivali s prozorom, web-forme možemo poistovjetiti s web-stranicom odnosno, kako se to često u sučelju Visual Studija naziva, web-dokumentom.

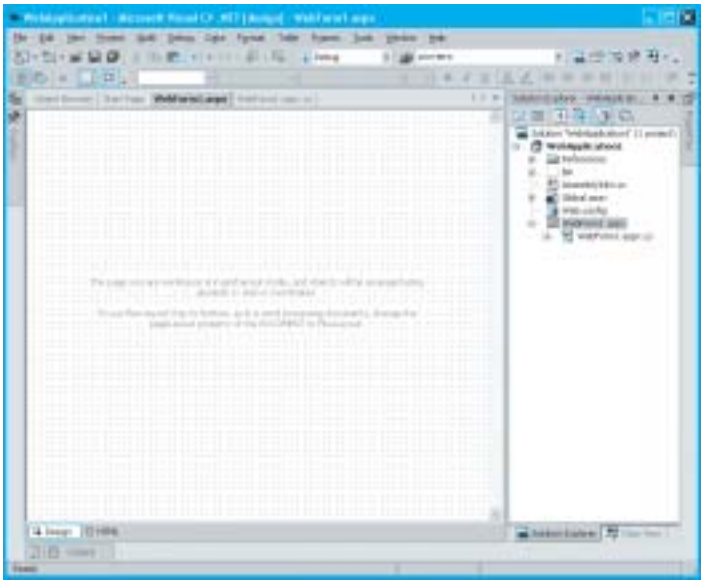
Web-aplikacije imaju jednu specifičnost – svaka je web-forma opisana s dvije datoteke. Prva datoteka sadrži sučelje opisano HTML-om i pojačano serverskim kontrolama (o tome nešto kasnije), dok u drugu datoteku dolazi kôd, funkcionalnost forme pisana u odabranom programskom jeziku.

To je princip koji nalaže Visual Studio, no nije jedini mogući. Naime, postoji mogućnost da oba dijela stranice stavite u istu datoteku, kao što to rade neki drugi razvojni alati, poput Web Matrixa. Međutim, odvajanje sučelja i kôda donosi puno prednosti, posebno na području preglednosti i čitljivosti. Takav način rada naziva se *code-behind*, dok se trpanje sučelja i kôda u istu datoteku zove *in-page coding*. Usporedbe radi, stari je ASP koristio ovaj drugi pristup što se često pokazivalo nepraktičnim.

Kod pisanja aplikacija u načinu *code-behind* datoteka sučelja ima nastavak ".aspx", dok datoteka s kôdom završava na ".aspx.cs". Drugi nastavak ovisi o programskom jeziku u kojem radimo – da koristimo Visual Basic .NET, nastavak bi bio ".aspx.vb".

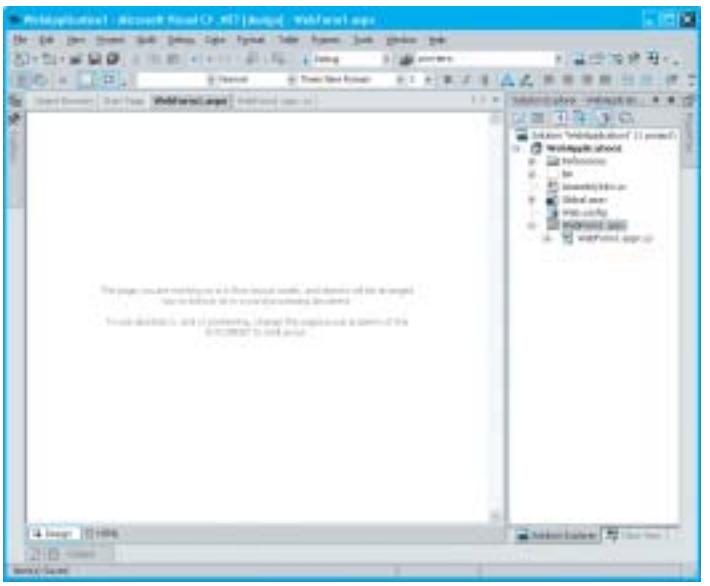
Sukladno upravo rečenom, na stranicu u sučelju Visual Studija možemo gledati na tri načina. Prvi, dizajnerski način (slike 10-3 i 10-4), omogućava nam izradu sučelja dovlačenjem kontrola iz prozora Toolbox. On može raditi u dva načina – GridLayout i FlowLayout. Prvi nam omogućava slaganje kontrola po volji – svaka od njih bit će pozicionirana na stranici na apsolutnim koordinatama, baš kao što smo to radili s prozorskim aplikacijama. Međutim, webu je puno prirodniji način ovaj drugi, FlowLayout, koji ne dozvoljava apsolutno pozicioniranje kontrola, već ih na stranicu smješta po redu, jednu za drugom, dok se, želimo li razmake među njima, moramo snalaziti tablicama, razmacima i prelascima u novi red. Za to će vam pak biti potrebno poznavanje jezika HTML, u čije tajne nemamo namjeru ulaziti u ovoj knjizi. Za tu tematiku preporučujemo knjigu *Izrada weba – abeceda za webmastere,* izdanu u istoj biblioteci.

Slika 10-3: ***Pogled na web-formu u dizajnerskom načinu – GridLayout***



Načini GridLayout i FlowLayout u dizajnerskom se načinu mogu razlikovati po točkicama – GridLayout u pozadini forme prikazuje mrežu točkica, dok je pozadina u načinu FlowLayout jednobojna.

Slika 10-4: ***Pogled na web-formu u dizajnerskom načinu – FlowLayout***



Iako vam se GridLayout vjerojatno čini privlačniji, aludiramo da ga ne koristite, posebno u projektima namijenjenima široj populaciji. Naime, on podrazumijeva neke stvari koje su na webu vrlo upitne, poput kompatibilnosti internetskih preglednika, fiksne dimenzije stranice i slično. Njih eventualno možete koristiti za izradu internih web-stranica namijenjenih manjoj, kontroliranoj skupini ljudi.

I sami ćemo poslušati vlastiti savjet – u primjerima ćemo se baviti isključivo načinom FlowLayout, pa stoga odmah na početku svojstvo pageLayout objekta Document postavite na tu vrijednost.

ASP.NET pripada u serverske tehnologije, što znači da se izvršava na strani servera, dok klijent dobiva kôd pisan u HTML-u. Ipak, i na serverskoj strani glavnu ulogu ima HTML – on i dalje predstavlja osnovu web-stranice i na njega se nadograđuje funkcionalnost u ASP.NET-u.

Svi u jednoga, jedan za sve

U knjizi pratimo rad Visual Studija i, sukladno tome, koristimo dvije datoteke – HTML-stranicu i pozadinski kôd. Međutim, red je da pokažemo kako se i gdje pozadinski kôd treba staviti u slučajevima kada želimo da oba dijela budu u istoj datoteci:

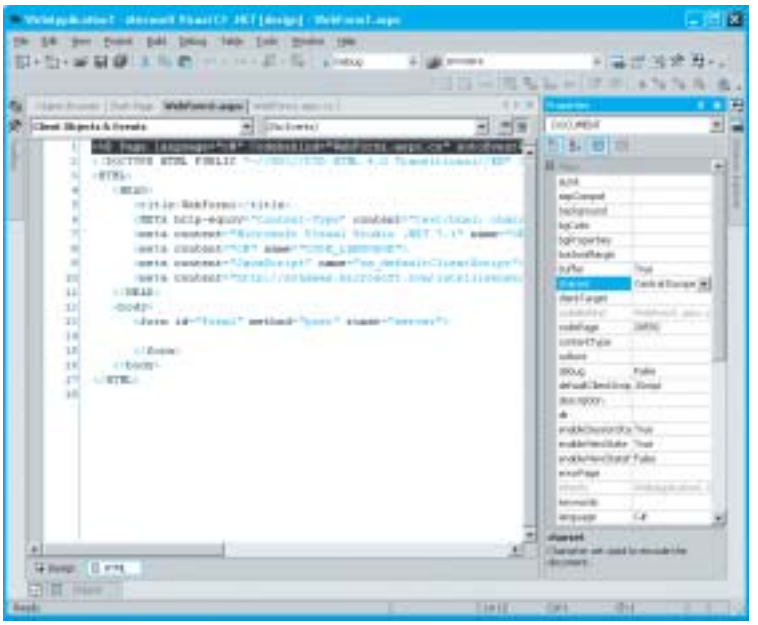
```
<%@ Page Language="C#" %>
<%@ Import
    Namespace="System.Data" %>

<html>
    <script runat="server">

        // tu ide "pozadinski" kôd

        void Page_Load(Object
            sender, EventArgs e)
          {
            // ...
        }
    </script>

    <body>
        <form runat="server">
        <!– tu idu HTML i
            serverske kontrole –>
        </form>
    </body>
</html>
```

Nekoć se to radilo na način da se kôd pisan u HTML-u kombinira sa skriptom, no ASP.NET, zahvaljujući svojoj objektnoj orijentaciji, radi na drugi način. Trik je u tome da elemente HTML-a pretvaramo u kontrole te se njima pristupa kao i kontrolama prozorskih aplikacija.

To omogućava razdvajanje HTML-a i programskog kôda, kao i mogućnost vezanja uz događaje nad tim objektima, što u HTML-u inače ne postoji.

Osim što postojeće elemente HTML-a možemo pretvoriti u serverske kontrole, možemo koristiti i ASP.NET-ove web-kontrole koje nisu ništa drugo nego povezana skupina HTML-elemenata koji pružaju određenu funkcionalnost.

Naravno, ne moraju svi elementi HTML-a postati kontrole – oni koji se ne kreiraju dinamički i dalje će biti obični elementi HTML-a bez ikakvih dodira sa serverskim skriptiranjem.



Slika 10-5: ***Pogled u HTML-kôd prazne stranice pisane u ASP.NET-u***

Osim serverskih kontrola, web-stranica koja zaslužuje nastavak .aspx mora imati i tzv. *Page*-direktivu koju ćete u sučelju Visual Studija prepoznati po žutoj boji. U njoj se definiraju određeni parametri izvršavanja skripte. Njih, dakako, ne morate pisati ručno, već možete iskoristiti sučelje Visual Studija – spomenuti parametri prikazani su kao svojstva objekta Document. Osim tih parametara, preko svojstava tog objekta utječemo i na neke karakteristike samog dokumenta, njegova zaglavlja (*head*) i osnovnih postavki tijela (*body*). Na direktive ćemo se vratiti nešto kasnije.

Serverske kontrole HTML-a

Dovucimo na web-formu iz sekcije HTML prozora Toolbox kontrolu Label. Ako nakon toga pogledamo u HTML-kôd, uočit ćemo da je dodan otprilike sljedeći kôd:

```
<DIV style="DISPLAY: inline; WIDTH: 70px; HEIGHT: 15px"
ms_positioning="FlowLayout">Label</DIV>
```

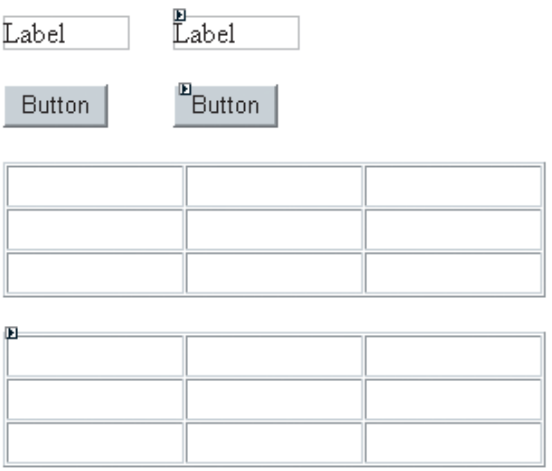
Unatoč određenim stilskim atributima i nestandardnim atributima *taga* Div, radi se o običnom HTML--elementu. Da bismo od njega dobili serversku kontrolu, moramo u dizajnerskom načinu označiti element, kliknuti desnom tipkom miša i u padajućem izborniku izabrati "Run as Server Control". Tim postupkom element će dobiti dva nova parametra, a cijeli će redak izgledati ovako:

```
<DIV style="DISPLAY: inline; WIDTH: 70px; HEIGHT: 15px" ms_positioning="FlowLayout"
id="DIV1" runat="server">Label</DIV>
```

HTML-element nije moguće promovirati u kontrolu pukim dodavanjem spomenutih parametara. Opisana radnja promocije manifestira se na još jednom mjestu, što ćemo vidjeti nešto kasnije.

Bilo kako bilo, nakon promoviranja elementa u serversku kontrolu, njemu i njegovim svojstvima možete pristupati iz pozadinskog kôda (koji se nalazi u drugoj datoteci). Pristupa mu se kao i svakom drugom objektu – navođenjem njegova imena. Primjerice ovako:

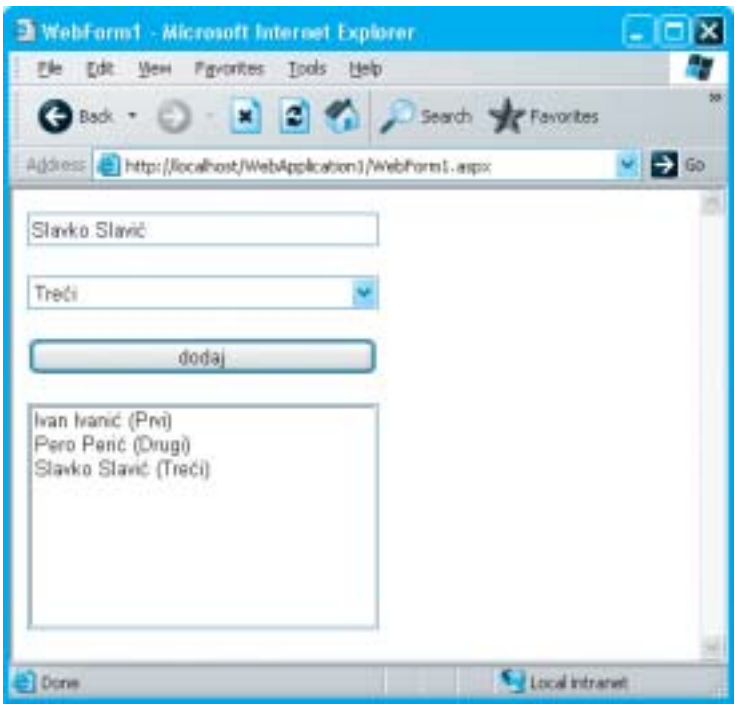
```
DIV1.InnerText = "Novi tekst za DIV1";
```



Slika 10-6: ***Serverske kontrole u dizajnerskom načinu razlikuju se po sitnim trokutićima.***

Primjer: Beskorisna aplikacija

No kako ne bi sve ostalo na pustoj teoriji, krenimo s konkretnim primjerom. I ovoga ćemo se puta poslužiti idejom beskorisne aplikacije jer je to zgodan način za upoznavanje osnovnih funkcija i jer ćete uočiti paralele s razvojem prozorskih aplikacija.



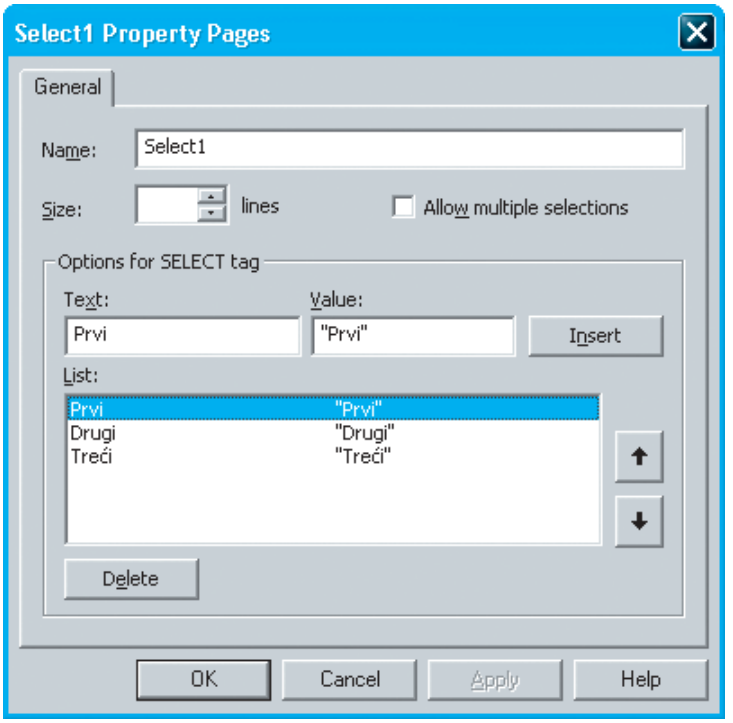
Slika 10-7: ***Beskorisna aplikacija u svom web-izdanju***

U tu ćemo svrhu na novootvorenu web-formu dovući potrebne elemente: po jedan Text Field, Drop-down, Button i Listbox. Sve ćemo promovirati u serverske kontrole i ostaviti im automatski dodijeljena imena.

Zatim ćemo kliknuti desnom tipkom miša na kontrolu DrowDown kako bismo pomoću prozora koji se krije iza stavke Properties dodali stavke padajućeg izbornika.

Nakon promjena, u pogledu na HTML stvar će izgledati ovako:

```
<SELECT id="Select1" name="Select1" runat="server">
   <OPTION value="Prvi">Prvi</OPTION>
   <OPTION value="Drugi">Drugi</OPTION>
   <OPTION value="Treći">Treći</OPTION>
</SELECT>
```



Slika 10-8: ***Pomoćni prozor za dodavanje stavaka padajućem izborniku***

Uočite kako neke različite kontrole koriste isti HTML-element za prikaz na stranici.

Preostaje nam još isprogramirati funkcionalnost. Za one koji se ne sjećaju, cilj je da se pritiskom na gumb uzmu vrijednosti iz tekstualnog polja i padajuće liste i zapišu na listu koja se nalazi ispod gumba (vidi sliku 10-7).

Oni koji su se bavili razvojem web-aplikacija znaju koliko je to prije ASP.NET-a bilo složeno, no sada se to svodi na jednu liniju kôda!

Dvokliknite na gumb u dizajnerskom načinu i otvorit će se funkcija koja će biti pozvana kada gumb bude pritisnut. U nju ćemo ubaciti sljedeći kôd:

```
private void Button1_ServerClick(...)
{
    Select2.Items.Add(Text1.Value + " (" + Select1.Value + ")");
}
```

Ukoliko vam se stvar čini poznata – u pravu ste! Postoje, doduše, određene razlike u imenima svojstava, no princip je u potpunosti jednak onome kod prozorskih aplikacija!

Proučimo što se događa u pozadini. Zavirimo li u HTML-kôd, možemo uočiti kako je cijelo tijelo stranice obuhvaćeno formularom – *tagovima* <form> i </form>. Taj formular nismo sami kreirali, već je on automatski ubačen prilikom kreiranja nove stranice. Kada kliknemo na gumb, sadržaj formulara šalje se serveru (kao i kod klasičnih web-stranica) i server vraća istu stranicu s izmijenjenim sadržajem. Cijela priča oko pamćenja stanja stranice obavlja se interno i programer na nju ne mora trošiti misli.

Ne treba posebno naglašavati da je stranica koja dolazi posjetitelju kao i svaka druga, s izuzetkom skrivenog parametra "__VIEWSTATE" koji pamti stanje stranice i njezine postavke. On je zaslužan za jednostavnost pisanja funkcionalnosti poput ove koju smo upravo pokazali. Pogledamo li iz drugog kuta, vidimo da Viewstate brine o vrijednostima serverskih kontrola odnosno da svaki element za koji želimo da Viewstate prati stanje moramo promovirati u serversku kontrolu.

Tablica 10-1: ***Lista serverskih kontrola i popis elemenata koji se u njih promoviraju***

HTML-element	Ime kontrole
HtmlAnchor	<a>
HtmlButton	<button>
HtmlForm	<form>
HtmlImage	<img>
HtmlInputButton	<input type="button">, <input type="submit">, <input type= "reset">
HtmlInputCheckBox	<input type= "checkbox">
HtmlInputFile	<input type="file">
HtmlInputHidden	<input type="hidden">
HtmlInputImage	<input type="image">
HtmlInputRadioButton	<input type="radio">
HtmlInputText	<input type="text">, <input type="password">
HtmlSelect	<select>
HtmlTable	<table>
HtmlTableCell	<td>, <th>
HtmlTableRow	<tr>
HtmlTextArea	<textarea>
HtmlGenericControl	Svi ostali elementi

Svaki HTML-element može biti promoviran u kontrolu, no pravo je pitanje – u koji tip kontrole će biti promoviran. Sve dostupne kontrole smještene su u klasi System.Web.UI.HtmlControls – za neke elemente postoje posebne kontrole, dok se ostali promoviraju u generičku kontrolu.

U koju se element kontrolu pretvorio možete vidjeti u pozadinskom kodu. Za svaku serversku kontrolu postoji jedan redak:

```
protected System.Web.UI.HtmlControls.HtmlInputText Text1;
protected System.Web.UI.HtmlControls.HtmlSelect Select1;
protected System.Web.UI.HtmlControls.HtmlInputButton Button1;
protected System.Web.UI.HtmlControls.HtmlSelect Select2;
```

Naime, kada promoviramo element u kontrolu, Visual Studio automatski dodaje i ovaj redak kako bismo mu mogli pristupati iz pozadinskog kôda. Drugim riječima, želite li neki element pretvoriti u kontrolu ručno, osim dodavanja prije spomenutih atributa, morat ćete dodati i deklaraciju poput ovih iz primjera. U izboru adekvatnog tipa kontrole pomoći će vam tablica 10-1.

Serverske kontrole u pravilu trebaju biti unutar glavnog formulara (dakle, između *tagova* <form> i </form>). Međutim, u serversku kontrolu možete promovirati i elemente izvan njega (primjerice, naslov stranice sadržan između *tagova* <title> i </title>). Ipak, kako su te kontrole izvan obrasca, bit će zakinute za neke mogućnosti.

Svaki tip serverske kontrole ima svoje karakteristike. Mi se njima, međutim, ovdje nećemo detaljno baviti (za samostalno proučavanje preporučujemo MSDN Library na putanji .NET Development > .NET Framwework SDK > .NET Framwework > Reference > Class Library > System.Web.UI.HtmlControls), već ćemo proučiti tek neke osnovne karakteristike.

Generička serverska kontrola

Bavit ćemo se generičkom serverskom kontrolom nazvanom HtmlGenericControl. Nju ćemo koristiti za sve elemente koji nemaju "svoju" serversku kontrolu. Međutim, sve stvari koje ovdje spomenemo vrijedit će i za ostale tipove serverskih kontrola.

Za primjer ćemo koristiti jednu kontrolu tipa Label. Jezikom HTML-a, radi se o običnom DIV elementu koji je automatski dobio ime DIV1. Nakon dovlačenja Labela i promoviranja u serversku kontrolu, njegov kôd će izgledati ovako:

```
<DIV style="DISPLAY: inline; WIDTH: 70px; HEIGHT: 15px" ms_positioning="FlowLayout"
id="DIV1" runat="server">Label</DIV>
```

Želimo li iz pozadinskog kôda izmijeniti tekst koji se u Labelu nalazi, napisat ćemo sljedeći izraz:

```
DIV1.InnerText = "Novi tekst";
```

No to smo već znali. Važno je napomenuti da spomenuti izraz moramo smjestiti u neku funkciju. Želimo li da se spomenuta promjena teksta dogodi prilikom učitavanja stranice, smjestit ćemo je u funkciju Page_Load koja je vezana upravo uz taj događaj. Naravno, postoji još hrpa događaja uz koju možemo vezati funkcije s određenom funkcionalnošću.

Ukoliko želimo u Label smjestiti neki izraz u HTML-u, koristit ćemo sljedeći izraz:

```
DIV1.InnerHtml = "<a href=\"http://www.bug.hr/\">click here</a>";
```

Uočite kako na mjesto navodnika stavljamo izraz \" – radi se o već spominjanom *escape*-znaku koji omogućava korištenje navodnika koji bi inače značili kraj znakovnog niza.

Želimo li promijeniti stil (CSS) kojim je Label ispisan, koristimo sljedeću sintaksu:

```
DIV1.Style["color"] = "Red";
DIV1.Style["font-weight"] = "bold";
```

To će na stranici rezultirati sljedećim izrazom:

```
<DIV style="color: Red; font-weight: bold;" ... >
```

Ukoliko želimo pristupiti nekom drugom atributu različitom od Style, primjerice svojstvu Align, napisat ćemo sljedeći izraz:

```
DIV1.Attributes["align"] = "right";
```

Na taj način možemo pristupati svim atributima serverskih kontrola, bez obzira na to o kojoj se kontroli radi. Naravno, neke kontrole imaju svoja, specifična svojstva koja nam olakšavaju rad s njihovim karakteristikama. Tako, primjerice, kontrola HtmlAnchor ima svojstvo HRef kojim direktno možemo pristupiti istoimenom atributu, a kontrola Button, primjerice, ima događaj koji nastupa kada se na nju klikne.

Sve vrijednosti svojstava serverskih kontrola HTML-a su tipa *string*.

Kao što ste mogli vidjeti iz prošlih primjera, serverske kontrole HTML-a prilično su nespretne za korištenje i brojne njihove funkcije nisu lako dostupne kroz sučelje Visual Studija. One su namijenjene naprednijim korisnicima – onima koji preferiraju pisanje vlastitih klijentskih skripti i žele imati potpunu kontrolu nad odgovorima web-poslužitelja. Za jednostavnije i objektnije programiranje tu su ASP.NET-ove web-kontrole.

ASP.NET-ove web-kontrole

Ove se kontrole nalaze u skupini Web Forms prozora Toolbox i puno su bolje podržane od strane Visual Studija. Zapravo, serverske kontrole HTML-a postoje radi kompatibilnosti, lakše nadogradnje i zahtjevnijih korisnika koji žele potpunu kontrolu – ne samo da se sve može riješiti pomoću web-kontrola, već to autori ASP.NET-a i očekuju. Zato ćemo se tim kontrolama nešto više posvetiti na stranicama što slijede.

Dovučemo li web-kontrolu na web-formu i nakon toga zavirimo u prikaz HTML-a, primijetit ćemo da se ne koriste standardni *tagovi* HTML-a, već neki posebni koji izgledaju otprilike ovako:

```
<asp:Label id="Label1" runat="server">Label</asp:Label>
```

Uočit ćete prefiks “asp:” koji ćemo pronaći na svim ugrađenim web-kontrolama. Osim tog prefiksa, kao što ćemo vidjeti kasnije, mogu postojati i drugi prefiksi za kontrole koje smo sami razvili ili preuzeli s Interneta.

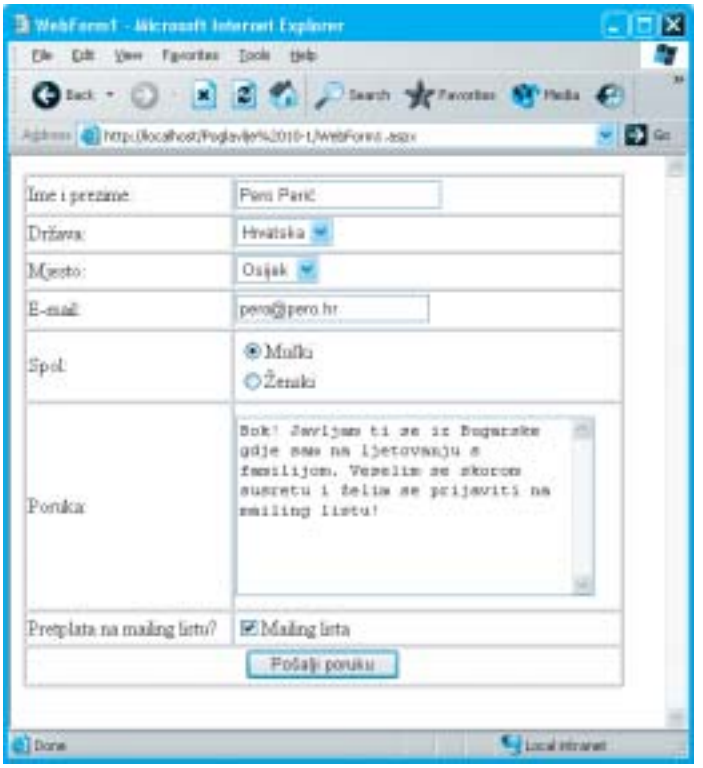
Treba odmah jasno reći da je ovo serverski prikaz kôda – kad stranica bude procesirana, klijent će dobiti klasičan kôd u HTML-u i to, što je najbolje, prilagođen pregledniku koji koristi.

Primjer: Obrazac za kontakt

Kako ne bi sve ostalo na pukoj teoriji, poslužit ćemo se primjerom. Ovoga puta ćemo izrađivati obrazac za kontakt u kojem će posjetitelj morati upisati određen broj podataka, koji će se nakon toga poslati na određenu e-mail adresu.

Za početak ćemo na formu smjestiti web-kontrole koje nam za primjer trebaju. Prvo ćemo postaviti jednu običnu tablicu (iz skupine HTML), čisto iz estetskih razloga. U prvi stupac ručno ćemo utipkivati natpise, dok ćemo u drugi postavljati adekvatne kontrole.

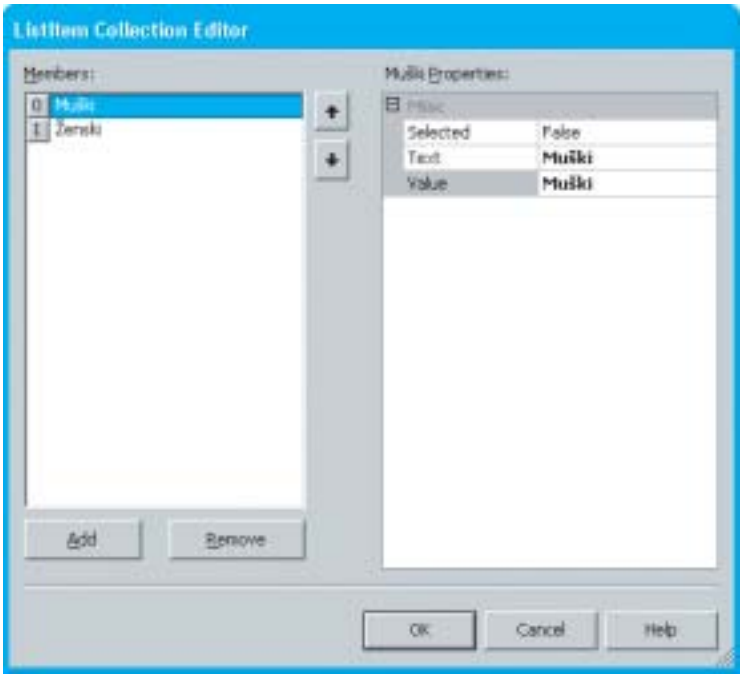
U prvom redu tako postavljamo kontrolu tipa TextBox – tekstualno polje za unos. Ono može biti jednolinijsko (svojstvo TextMode postavljeno na SingleLine), višelinijsko (MultiLine) ili sadržavati lozinku (Password). Osim tog svojstva pronaći ćemo i brojna druga svojstva. Od vizualnih treba izdvojiti svojstvo CssClass kojim kontroli pridružujemo klasu stilskog predloška, a od ostalih spomenut ćemo svojstvo Text u kojem se nalazi znakovni niz upisan u kontrolu. Ostala svojstva prepoznat ćete i sami po njihovu imenu, a jednu skupinu svojstava ostavit ćemo za kasnije. Za potrebe primjera kontrolu smo preimenovali u ImePrezimeText.



Slika 10-9:
Kako će primjer Obrazac za kontakt izgledati u konačnici

U drugom i trećem retku postavljamo dvije kontrole tipa DropDownList, koje ćemo nazvati DrzavaDDL i GradDDL. I tu nema novih i nepoznatih svojstava – možda samo treba napomenuti da se stavke tog padajućeg izbornika mogu definirati preko kolekcije Items. Mi to u ovom primjeru nećemo učiniti na taj način, no imajte na umu da je stvar vrlo slična prozorskom padajućem izborniku.

Slijedi još jedno tekstualno polje za upis (nazvano EMailText), a u petom ćemo redu susresti kontrolu tipa RadioButtonList (nazvanu SpolRadioList). Ona nije ništa drugo nego skupina *radiobuttona* s kojom se radi na isti način kao i s kontrolom DropDownList. Izbore ćemo definirati preko svojstva Items, a odabranoj vrijednosti, kao što ćemo kasnije vidjeti, pristupat ćemo na potpuno isti način. Usto, ova kontrola krije neka zanimljiva svojstva. Naime, ASP.NET nam pruža mogućnost odluke kako će ponuđene opcije biti prikazane. Želimo li umjesto predefiniranog nabrajanja izbora jedan ispod drugoga odrediti da budu smješteni u istom redu, mijenjat ćemo svojstvo RepeatDirection. Izbori mogu biti smješteni u posebnoj, nevidljivoj tablici (što izgleda ljepše) ili izvan nje. To odlučujemo svojstvom RepeatLayout. Konačno, ako izbora ima toliko da nam ne paše niti vodoravni niti okomiti raspored, možemo ih posložiti u tablicu s više stupaca i redova, što definiramo svojstvom RepeatColumns.



Slika 10-10:
Pomoćni prozor za unos izbora u kontrolu tipa RadioButtonList

U redu za pisanje poruke ubacili smo još jedno tekstualno polje, nazvali ga PorukaText, no ovaj put smo mu svojstvo TextMode postavili na vrijednost MultiLine.

Slika 10-11:
Modaliteti svojstava RepeatDirection i RepeatColumns

```
RepeatDirection = RepeatDirection.Vertical;
RepeatColums = 0;
```

Prvi
Drugi
Treći
Četvrti
Peti
Šesti
Sedmi
Osmi

```
RepeatDirection = RepeatDirection.Horizontal;
RepeatColums = 0;
```

Prvi Drugi Treći Četvrti Peti Šesti Sedmi Osmi

Prvi Drugi Treći
Četvrti Peti Šesti
Sedmi Osmi

Prvi Četvrti Sedmi
Drugi Peti Osmi
Treći Šesti

```
RepeatDirection = RepeatDirection.Vertical;
RepeatColums = 3;
```

```
RepeatDirection = RepeatDirection.Horizontal;
RepeatColums = 3;
```

Na kraju dolazi obična kontrola tipa CheckBox, koju smo nazvali MailingCheck. U svojstvo Text valja staviti tekst koji će se iza nje pojaviti i na koji će ona biti vezana dok se uključenost odnosno isključenost može iščitavati svojstvom Checked, isto kao i kod prozorskih aplikacija.

Na kraju, na dnu obrasca, smještamo gumb – kontrolu tipa Button koja će služiti za slanje obrasca. Gumb smo nazvali SlanjeButton, a na tekst koji je na njemu upisali smo u svojstvo Text. Time smo zgotovili sučelje primjera.

Vraćanje stranice poslužitelju

Kao što smo već vidjeli kod serverskih kontrola HTML-a, cijela je stranica obrazac koji se vraća poslužitelju kada kliknemo na gumb. To vraćanje naziva se *postback* i ono, osim gumbom, može biti inicirano od strane bilo koje druge kontrole.

Objasnimo to primjerom. Nadogradit ćemo naš primjer tako da se, ovisno o izabranoj državi, popuni padajuća lista gradova. Za početak moramo napraviti neke pripremne radnje – napraviti dva polja s popisom država i gradova. Među deklaraciju varijabli (nakon deklaracije kontrola) dodajemo sljedeće linije:

```
...
protected System.Web.UI.WebControls.TextBox ImePrezimeText;

private string[] drzave = new string[4];
private string[][] gradovi = new string[4][];
```

Kreirana polja moramo popuniti podacima. U praksi bi se to vjerojatno čupalo iz neke baze podataka, no, jednostavnosti radi, mi ćemo koristiti direktnu populaciju na način koji slijedi. Kôd stavljamo u metodu vezanu uz događaj učitavanja stranice kako bi podaci bili od početka dostupni:

```
private void Page_Load(object sender, System.EventArgs e)
{
   // Put user code to initialize the page here
   drzave[0] = "Hrvatska";
   drzave[1] = "Italija";
   drzave[2] = "Portugal";
   drzave[3] = "Brazil";

   gradovi[0] = new string[] { "Zagreb", "Split", "Rijeka", "Osijek" };
   gradovi[1] = new string[] { "Rim", "Siena", "Milano" };
   gradovi[2] = new string[] { "Lisabon", "Porto", "Fatima" };
   gradovi[3] = new string[] { "Sao Paolo", "Rio de Janeiro", "Minas Gerais" };
```

Nakon toga, trebamo te podatke smjestiti u pripadajuće padajuće izbornike. To ćemo učiniti ovako:

```
    DrzavaDDL.DataSource = drzave;
    DrzavaDDL.DataBind();
    GradDDL.DataSource = gradovi[DrzavaDDL.SelectedIndex];
    GradDDL.DataBind();
}
```

Prva dva reda su jasna – prvom padajućem izborniku pridružujemo polje *drzave* kao DataSource, zatim te podatke "izvlačimo" u kontrolu metodom DataBind(). U drugom dijelu radimo istu stvar, no kako je polje *gradovi* dvodimenzionalno, radimo to samo s onim retkom koji pripada izabranoj državi. Kako je prva država na popisu Hrvatska, ovim komadom kôda će se kontrola GradDDL popuniti gradovima iz Hrvatske.

E sad, što kada posjetitelj izmijeni odabranu državu? U tom slučaju kontrolu GradDDL treba popuniti drugim podacima, gradovima koji se nalaze u toj državi. Stoga se treba vezati na događaj SelectionIndexChanged kontrole DrzavaDLL. U tu ćemo funkciju upisati sljedeće:

```
private void DrzavaDDL_SelectedIndexChanged(...)
{
    GradDDL.DataSource = gradovi[DrzavaDDL.SelectedIndex];
    GradDDL.DataBind();
}
```

Stvar bi besprijekorno radila u prozorskoj aplikaciji, no kod web-aplikacija imamo još malo posla. Naime, kako je ASP.NET serverska tehnologija, znamo da, ako želimo da se nešto na formi promijeni, moramo vratiti stranicu serveru odnosno napraviti *postback*. To bismo u našem primjeru napravili pritiskom na gumb, no želimo da se promjena izvrši odmah nakon promjene selekcije. Stoga među svojstvima kontrole DrzavaDLL treba pronaći AutoPostBack i postaviti ga na *true*. Na taj način smo stranici dali naredbu da promjena izbora u toj kontroli pošalje obrazac serveru na procesiranje.

Sjetimo se, zahvaljujući pamćenju stanja pomoću skrivenog parametra "__ViewState", sustav će zapamtiti sve što smo dotada upisali pa, bez obzira na to što ponovo učitavamo stranicu, dotada upisane vrijednosti neće biti izgubljene.

Međutim, doći će do jedne druge neželjene nuspojave. Kada server vrati stranicu, ponovo će se izvršiti metoda Page_Load, čime će naše padajuće liste biti nanovo ispunjene. Tom radnjom izgubit će se učinjeni izbor pa cijela stvar uopće neće raditi kako smo očekivali.

Stoga spomenutoj metodi treba naložiti da punjenje padajućih izbornika napravi samo prilikom prvog učitavanja stranice. To ćemo napraviti pomoću svojstva Page.IsPostBack koji, ako ima vrijednost *true*, označava da je stranica vraćena (odnosno da je napravljen *postback*). Dakle, konačna verzija metode izgledat će ovako:

```
private void Page_Load(object sender, System.EventArgs e)
{
   drzave[0] = "Hrvatska";
   drzave[1] = "Italija";
   drzave[2] = "Portugal";
   drzave[3] = "Brazil";
   gradovi[0] = new string[] { "Zagreb", "Split", "Rijeka", "Osijek" };
   gradovi[1] = new string[] { "Rim", "Siena", "Milano" };
   gradovi[2] = new string[] { "Lisabon", "Porto", "Fatima" };
   gradovi[3] = new string[] { "Sao Paolo", "Rio de Janeiro", "Minas Gerais" };

   if (!Page.IsPostBack)
   {
      DrzavaDDL.DataSource = drzave;
      DrzavaDDL.DataBind();
      GradDDL.DataSource = gradovi[DrzavaDDL.SelectedIndex];
      GradDDL.DataBind();
   }
}
```

Uočite da se punjenje podataka u polja nalazi izvan uvjetnog bloka i da će biti izvršeno prilikom svakog učitavanja stranice, bez obzira na to radi li se o vraćenoj stranici ili ne. Naime, pamćenje stanja kroz skriveni parametar vrijedi samo za kontrole, a ne i varijable koje definiramo u pozadinskom kodu.

Osim iniciranja slanja stranice serveru, kontrole možemo i isključiti iz pamćenja stanja. Drugim riječima, isključimo li kontroli svojstvo EnableViewState njezino stanje neće biti zapamćeno nakon vraćanja stranice serveru već će biti postavljeno na onu početnu, definiranu svojstvima u dizajnerskom načinu.

Želite li u potpunosti isključiti pamćenje stanja, jednostavno izbrišite obrazac u kojem se cijela stranica nalazi. Ipak, imajte na umu da ćete tim postupkom izgubiti velik dio funkcionalnosti.

Slanje elektroničke poruke

Pozabavimo se konačno slanjem *e-maila*. I sami pogađate – funkcionalnost ćemo vezati uz događaj klika na gumb. U pripadajuću metodu stavljamo sljedeće:

```
private void SlanjeButton_Click(object sender, System.EventArgs e)
{
    string Poruka = "";
    Poruka += "Ime i prezime: " + ImePrezimeText.Text + Environment.NewLine;
    Poruka += "Država: " + DrzavaDDL.SelectedValue + Environment.NewLine;
    Poruka += "Grad: " + GradDDL.SelectedValue + Environment.NewLine;
    Poruka += "Spol: " + SpolRadioList.SelectedValue + Environment.NewLine;
    Poruka += "Pretplata na mailing listu? " + (ListaCheck.Checked  ? "Da" : "Ne")
      + Environment.NewLine;
    Poruka += Environment.NewLine;
    Poruka += PorukaText.Text + Environment.NewLine;

    SmtpMail.SmtpServer = "localhost";
    SmtpMail.Send(EMailText.Text, "primatelj@server.hr", "Poruka s Interneta",
      Poruka);
}
```

U prvih nekoliko redaka stvaramo poruku koja će biti poslana. Mogli smo sve smjestiti u isti redak, no ovako je preglednije. U tom dijelu možete vidjeti i kako se pristupa upisanim odnosno izabranim vrijednostima kontrola. Za prelazak u novi redak poruke koristimo konstantu Environment.NewLine.

Slanje poruke je izuzetno jednostavno i može se izvesti pomoću jedne linije. Ipak, prije korištenja moramo na početak datoteke dodati referencu na pripadajući *namespace*:

```
using System.Web.Mail;
```

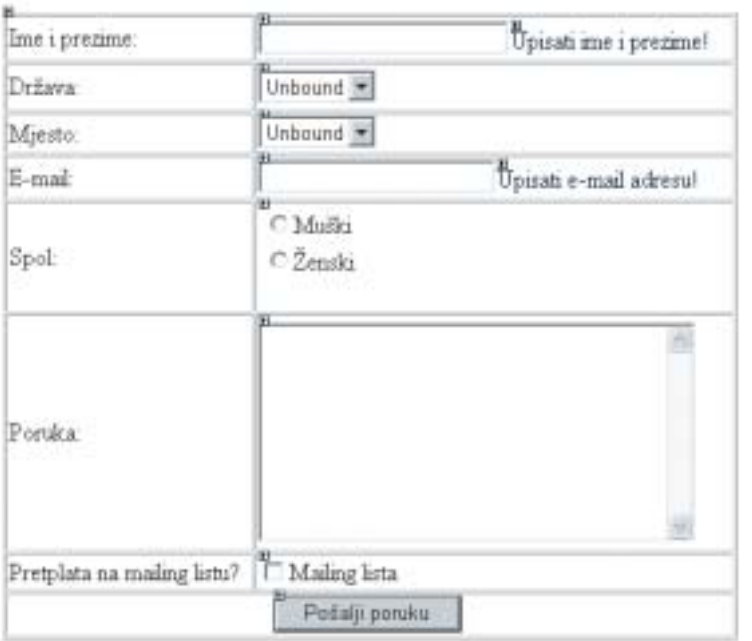
Poruku šaljemo pomoću metode Send, kojoj kao parametre navodimo adresu pošiljatelja, adresu primatelja, temu poruke i samu poruku. Mi smo u primjer dopisali i definiranje svojstva SmtpServer, kako biste u svojim primjerima mogli koristiti i neki drugi od *defaultno* postavljenog.

Ovo je tek najjednostavniji oblik slanja poruke. Osim na ovaj način, poruke je moguće slati i na druge načine, koristeći druge metode i svojstva. Detalje potražite u dokumentaciji – ime *namespacea* znate.

Provjera valjanosti upisanog

Iskusniji webaši primijetit će da stvar još nije niti približno gotova. Što, primjerice, ako posjetitelj ne upiše svoju e-mail adresu i tako pokuša poslati poruku? Ili, što ako uopće ne ispuni obrazac, nego odmah klikne na gumb? Osim što od takvog praznog obrasca ne bi bilo nikakve koristi, u našem bi se slučaju javila i greška jer nije dozvoljeno poslati poruku bez adrese pošiljatelja, koja se vadi direktno iz kontrole EMailText. Drugim riječima, trebamo uvesti provjeru valjanosti upisanog ili, stručnim rječnikom, validaciju.

Validacija se, osim ukoliko drugačije ne postavimo, radi korištenjem klijentskog skriptiranja. Naravno, vi o tome ne morate brinuti – stvar je u potpunosti riješena unutar samih web-kontrola. Za posjetitelje s preglednicima koji ne podržavaju klijentske skripte, validacija će se vršiti na serverskoj strani.



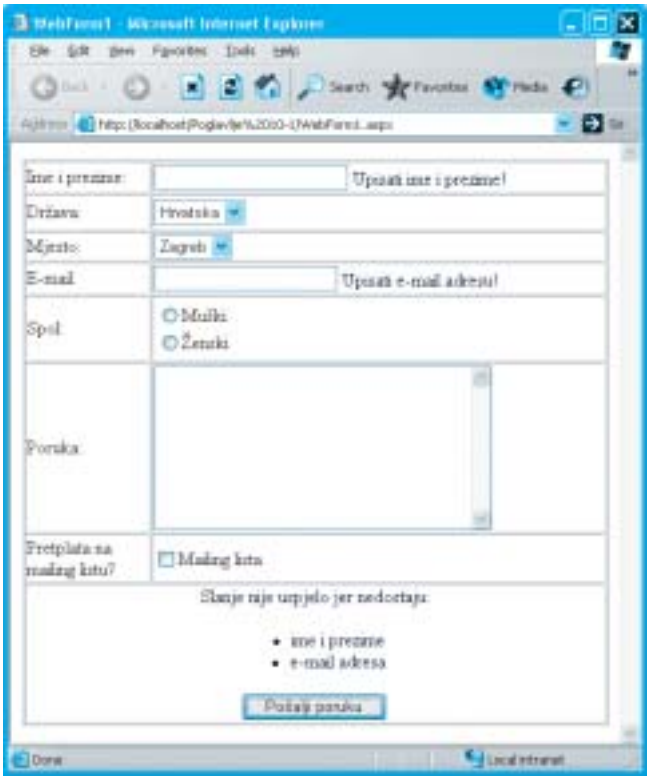
Slika 10-12: ***Validatori dodani na kontrole ImePrezimeText i EMailText***

Dovucimo stoga dvije kontrole tipa RequiredFieldValidator i smjestimo ih kraj pripadajućih kontrola (vidi sliku 10-12). Imajte na umu da nije nužno da validacijske kontrole budu blizu onih na

koje su vezane, no tako je vizualno logičnije. Povezivanje validacijske kontrole s onom koju provjerava radimo preko svojstva ControlToValidate. U pripadajućem padajućem izborniku pojavit će se popis svih trenutno postojećih kontrola na web-formi, pa je dovoljno izabrati. Nakon toga, ta će se kontrola brinuti o tome da ne omogućava slanje obrasca a da nešto nije upisano u polje na koje je vezana.

Prema osnovnim postavkama validacijskih kontrola procedura će izgledati ovako: ako kliknemo na gumb a da nismo ništa unijeli u kontrole koje se provjeravaju, na stranici će se pojaviti dotada skrivene validacijske kontrole. Tekst koji želimo da prikažu u tom trenutku treba upisati u njihovo svojstvo Text.

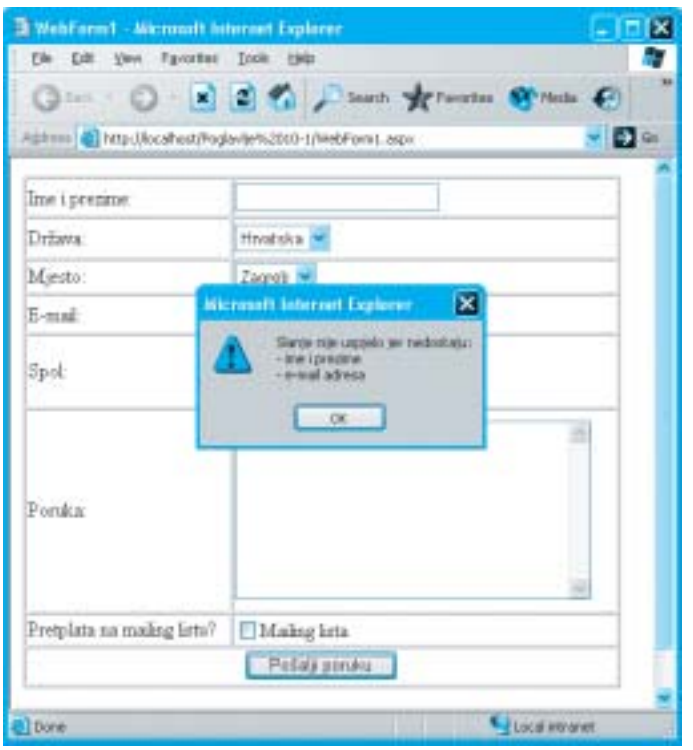
Iako se ne vide, validacijske kontrole zauzimaju određeno mjesto na stranici. To je često neprihvatljivo ponašanje. Srećom, moguće ga je isključiti. To činimo tako da svojstvo Display iz Static promijenimo u Dynamic. Na taj način kontrola će zauzimati prostor na stranici samo kada bude vidljiva.



Slika 10-13: ***Poruke validacijskih kontrola prikazane na oba mjesta***

Ponekad je nezgodno da se upozorenje o neispunjenom polju javlja kraj njega. U tim slučajevima bismo radije da se poruke svih validacijskih polja prikazuju na istom, zajedničkom mjestu. Da bismo to postigli, dovući ćemo kontrolu tipa ValidationSummary. Samim time odredili smo zajedničko

mjesto na kojem će se pojavljivati upozorenja svih validacijskih kontrola na stranici. Štoviše, svaka će se greška manifestirati na dva mjesta – na mjestu svoje validacijske kontrole i na zajedničkom prostoru. Te dvije poruke ne moraju biti identične – prvu, kao što smo već spomenuli, definiramo svojstvom Text, dok se na zajedničkom mjestu pokazuje ona smještena u svojstvo ErrorMessage. Želite li da se poruka pokazuje samo na zajedničkom dijelu, treba svim validacijskim kontrolama (osim ValidationSummary) svojstvo Display postaviti na vrijednost None.



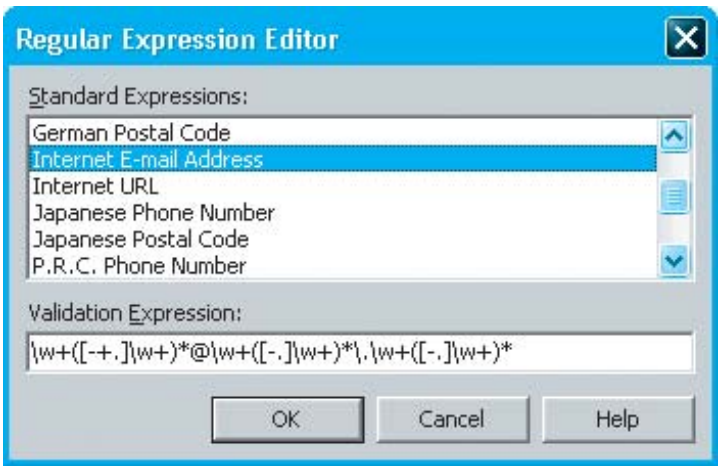
Slika 10-14: ***Iskakajuće upozorenje o neispunjenosti polja***

Po nama, najbolji način prikaza poruka o nepravilno ispunjenim poljima je upozorenje kakvo možete vidjeti na slici 10-14. S tim ciljem treba kontroli ValidationSummary uključiti svojstvo ShowMessageBox, a isključiti ShowSummary.

Naravno, moguće su i razne druge kombinacije spomenutih načina prikaza informacije o grešci.

U našem primjeru provjera postojanja bilo kakvog upisa za polje EMailText nije dovoljna. Naime, moramo posjetitelja upozoriti i u slučajevima kada upiše nevaljalu adresu elektroničke pošte. Za to će nam poslužiti kontrola RegularExpressionValidator koja provjerava zadovoljava li upisano određeni predložak.

Korištenje svih validacijskih kontrola je slično pa i kod ove treba proći istu proceduru vezanja uz kontrolu i definiranja načina ponašanja. Međutim, tu moramo definirati jedno svojstvo više – svojstvo ValidationExpression koje određuje predložak koji je dozvoljeno unijeti.



Slika 10-15: ***Pomoćni prozor za izbor validacijskog predloška***

Sintaksa predložaka prilično je složena. Srećom, postoji predefinirani predložak za ono što nam u primjeru treba pa je dovoljno kliknuti na gumbić spomenutog svojstva te u prozoru koji će se otvoriti odabrati izraz "Internet E-mail Address" (slika 10-15).

Nakon toga, ta validacijska kontrola posjetitelja neće dalje puštati sve dok ne unese pravilnu adresu elektroničke pošte.

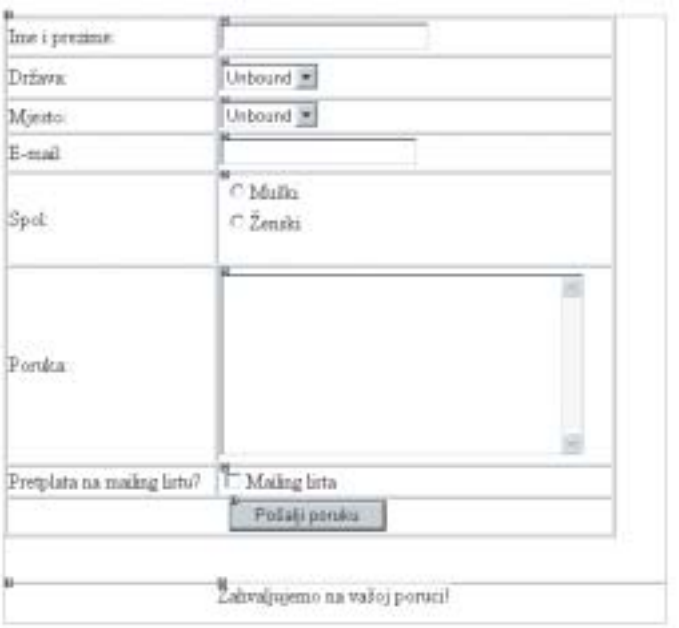
Osim spomenutih validacijskih kontrola postoje još neke koje ćemo tek telegrafski spomenuti. Prva među njima je CompareValidator, koja uspoređuje dvije kontrole (zgodno kod dvostrukog unosa lozinke) ili kontrolu uspoređuje s nekom predefiniranom vrijednošću. Isti se validator koristi i za insistiranje na određenom tipu unesenog podatka, primjerice datuma ili broja (svojstvo Operator postavlja se na DataTypeCheck, a Type na tip koji želimo tražiti).

Kontrola RangeValidator uspoređuje pripada li neka vrijednost određenom rasponu, dok nam kontrola CustomValidator omogućava samostalno programiranje validatora za koji ne postoji ugrađena kontrola.

I na kraju sage, otkrijmo što zapravo inicira proces provjere valjanosti. Radi se o svojstvu CausesValidation koji je u standardnoj postavi kontrola tipa Button (i još ponekih). Ukoliko ga isključimo, validacija neće nastupiti.

Paneli

Kad smo već tako blizu, dotjerajmo primjer do savršenstva (barem što se programerskog dijela tiče, s vizualnošću bi se još štošta dalo učiniti). Naime, nedostaje nam potvrda da je poruka poslana. Umjesto nje, nakon slanja će se pojaviti isti obrazac, pa bi se moglo dogoditi da posjetitelj niti ne primijeti da je slanje obavljeno.



Slika 10-16: ***Panele u dizajnerskom načinu prepoznajemo po tankom sivom obrubu koji se u pokrenutoj aplikaciji ne vidi.***

U tu ćemo svrhu na web-formu dovući dvije kontrole tipa Panel. Kao i kod prozorskih aplikacija, radi se o kontrolama koje nemaju vlastitu funkcionalnost, već sadrže druge kontrole. U prvu od njih ćemo smjestiti kompletno dosada napravljeno sučelje (jednostavno označite sve kontrole, desni klik na jednu od njih i iz padajućeg izbornika odaberite Cut, zatim desnom tipkom miša kliknite na Panel i odaberite Paste – imena i sva svojstva ostat će sačuvana). U drugi Panel ćemo smjestiti tekst zahvale. Panele nazovite ObrazacPanel i ZahvalaPanel, a potonjem svojstvo Visible postavite na *false,* kako se ne bi pokazivao na stranici.

Još nam preostaje napisati dvije linije kôda, na kraj funkcije vezane uz događaj klika na gumb:

```
ObrazacPanel.Visible = false;
ZahvalaPanel.Visible = true;
```

Nakon što poruka bude poslana, ObrazacPanel će se zajedno sa svim svojim kontrolama sakriti, a ZahvalaPanel bit će prikazan zajedno sa svojom rečenicom zahvale na ispunjenom obrascu.

Izrada korisničkih kontrola

Osim korištenja postojećih kontrola ili uključivanja tuđih, preuzetih s Interneta (vidi poglavlje o prozorskim aplikacijama), moguće je izrađivati i vlastite kontrole. Vaša kontrola može biti nadogradnja neke postojeće ili spajanje više različitih kontrola u neku logički povezanu funkcionalnost.

Vlastite kontrole koriste se kao i ugrađene, samo što, kao što ćemo kasnije vidjeti, imaju vlastiti prefiks. No krenimo redom...

Jednostavna web-kontrola

Prvo ćemo izraditi vrlo jednostavnu web-kontrolu koja će ispisivati IP-adresu posjetitelja. Postupak je sljedeći: u izborniku Project kliknite na stavku Add Web User Control. U prozoru upišite ime nove kontrole i potvrdite odabir klikom na OK.

Pokazat će se radna površina gotovo identična onoj koju smo vidjeli na početku izrade web-forme. U nju možete pisati tekst, stavljati HTML-elemente i koristiti serverske kontrole – sve što želite da vaša kontrola sadrži.

Mi ćemo za naš primjer otipkati tekst "Vaša IP-adresa:" te nakon njega staviti web-kontrolu Label. Zatim ćemo u pozadinskom kodu u metodu Page_Load upisati sljedeće:

```
private void Page_Load(object sender, System.EventArgs e)
{
    Label1.Text = Request.ServerVariables["REMOTE_ADDR"];
}
```

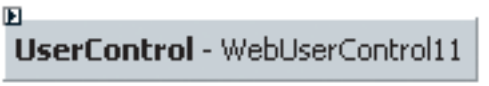
Tim kôdom ćemo u kontrolu Label1 upisati posjetiteljevu IP-adresu. Ona je zapisana u tzv. serverskoj varijabli REMOTE_ADDR kojoj pristupamo preko kolekcije Request.ServerVariables.

I to je to. Da bismo kontrolu pripremili za korištenje, trebamo napraviti *rebuild* cijelog projekta (jer naša je kontrola dio otvorenog projekta). Na taj način kontrola će se kompajlirati i moći ćemo je koristiti na svojoj web-formi. *Rebuild* radimo tako da desnom tipkom miša kliknemo na ime projekta u Solution Exploreru i iz izbornika izaberemo stavku Rebuild.

Korisničke kontrole imaju nastavak ".ascx", što je skraćeno od *active server control*.

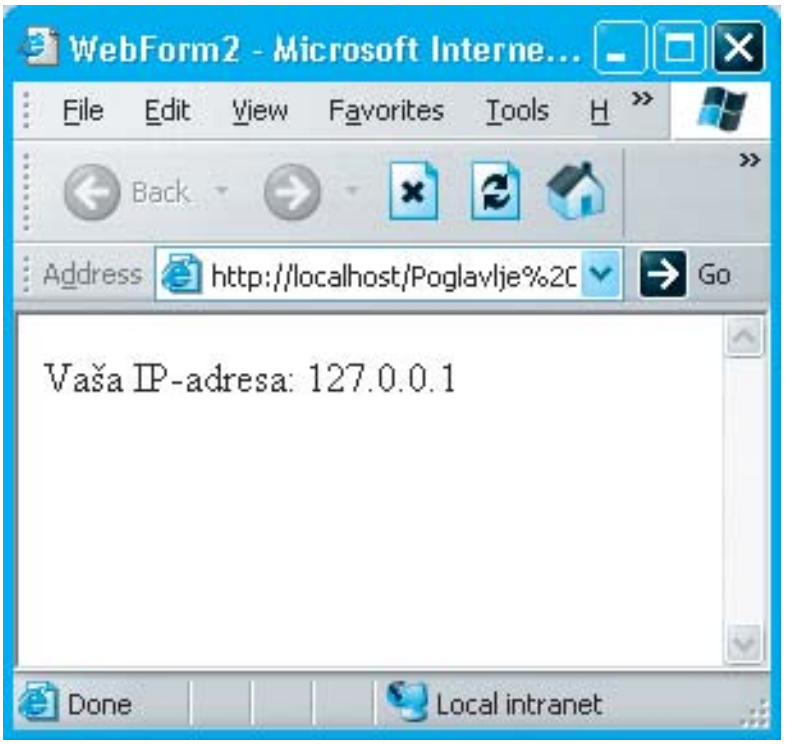
Zatim se trebamo vratiti na dizajnerski način web-forme i jednostavno iz Solution Explorera dovući napravljenu kontrolu.

Slika 10-17: ***Korisnička kontrola u dizajnerskom načinu izgleda ovako.***



U dizajnerskom načinu kontrola će izgledati kao na slici 10-17, no ako formu pokrenemo, ona će pokazati svoje pravo lice i funkcionalnost.

Slika 10-18: ***Kad otvorimo formu u pregledniku, naša će korisnička kontrola izgledati ovako.***



Pogledajmo što se dogodilo u pozadini. Otvorimo stoga web-formu tako da joj vidimo HTML-kôd. Odmah na početku otkrit ćemo sljedeću (žutu) liniju:

```
<%@ Register TagPrefix="uc1" TagName="WebUserControl1" Src="WebUserControl1.ascx" %>
```

Ta direktiva registrira našu komponentu za korištenje na web-formi. Parametrom TagPrefix određujemo njen prefiks, parametrom TagName ime *taga* koji ćemo koristiti, a Src označava datoteku u kojoj se kontrola nalazi.

Nešto kasnije u kodu pronaći ćemo sljedeću liniju:

```
<uc1:WebUserControl1 id="WebUserControl11" runat="server"></uc1:WebUserControl1>
```

Time pozivamo našu serversku kontrolu. Primijetite prefiks "uc1" i *tag* WebUserControl1 koji smo definirali na vrhu stranice. Da tamo promijenimo prefiks i *tag*, morali bismo ih promijeniti i ovdje.

Jednom registriranu kontrolu na vrhu stranice u kodu možemo pozivati proizvoljan broj puta. Bitno je samo svaki put dodijeliti joj drugo identifikacijsko ime (parametar ID).

Složena web-kontrola

Priznajte, ovo je bilo prelagano. Zato ćemo se sada poigrati s nešto složenijom kontrolom kojoj ćemo prosljeđivati određeni parametar.

Da bismo to postigli, trebamo se podsjetiti da je svaka kontrola zapravo klasa. Dodavanje parametara web-kontroli zapravo je dodavanje parametara klasi, što smo već naučili u šestom poglavlju.

Napravit ćemo kontrolu koja će biti specijalizirana za prikaz linka na e-mail adresu. Tu funkcionalnost, dakako, možemo imati i sada, tako da kontroli HyperLink u svojstvo NavigateUrl ubacimo e-mail adresu s prefiksom "mailto:". Međutim, našoj ćemo kontroli morati kao parametar dati samo e-mail adresu, a ona će se pobrinuti oko dodavanja prefiksa i postavljanja svojstva Text.

Otvorimo stoga novu korisničku kontrolu kao i u prošlom primjeru i dovucimo kontrolu tipa HyperLink. Zatim se prebacimo u pozadinski kôd i dodajmo sljedeće:

```
public string EMail
{
    set
    {
        HyperLink1.Text = value;
        HyperLink1.NavigateUrl = "mailto:" + value;
    }
    get
    {
        return HyperLink1.Text;
    }
}
```

Deklarirali smo, dakle, javnu varijablu (parametar) EMail. U bloku *set* definiramo što će biti kada se njezina vrijednost postavlja – bit će pridružena svojstvu Text kontrole HyperLink1 kao i svojstvu NavigateUrl, ali s ubačenim prefiksom.

U drugom bloku (*get)* definiramo da ukoliko pristupamo varijabli EMail bude vraćena vrijednost svojstva HyperLink1.Text pošto se u njemu nalazi ta vrijednost. Nismo mogli vratiti vrijednost svojstva HyperLink1.NavigateUrl jer on osim adrese sadrži i prefiks.

U slučaju da se niti jedno svojstvo ne podudara s vrijednošću javne varijable, trebamo uvesti dodatnu, internu varijablu za pamćenje te vrijednosti. Ovako:

```
private string EMailAdresa;
public string EMail
{
   set
   {
      EMailAdresa = value;
      HyperLink1.Text = value;
      HyperLink1.NavigateUrl = "mailto:" + value;
   }
   get
   {
      return EMailAdresa;
   }
}
```

Nakon kompajliranja, ovu ćemo kontrolu moći koristiti na web-formi. Samu kontrolu možemo dovući iz Solution Explorera, dok ćemo parametar ubaciti ručno, na sljedeći način:

```
<uc1:SlozenaKontrola id="SlozenaKontrola1" runat="server"
EMail="pero@pero.hr"></uc1:SlozenaKontrola>
```

Web-aplikacija

Sve što smo dosad spomenuli odnosilo se na jednu skriptu. ASP.NET, međutim, poznaje pojam aplikacije koja, pojednostavljeno rečeno, čini skup više skripti u istoj mapi i pripadajućim podmapama.

Inicijalno, svaki web-*site* definiran u IIS-u je zasebna web-aplikacija. Međutim, zasebnim aplikacijama možete proglasiti i mape unutar *sitea*. Sjetimo se da je Visual Studio kod kreiranja novog projekta stvorio virtualnu mapu koja je ujedno i samostalna web-aplikacija. Da biste to učinili sami, potrebno je ući u konzolu za konfiguraciju IIS-a (nalazi se u Control Panel > Administrative

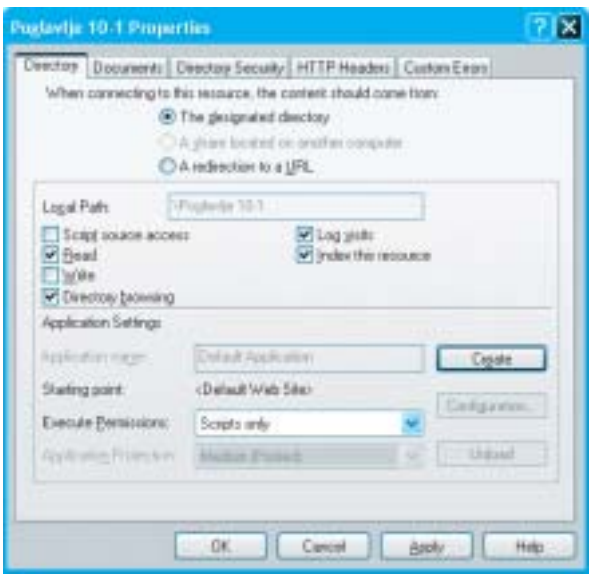
ASP.NET-ove mobilne kontrole

O sim izrade web-stranica za "normalne" preglednike, .NET Framework omogućava izradu tzv. mobilnih web-stranica korištenjem ASP.NET-ovih mobilnih kontrola. Radi se o kontrolama koje su svojim karakteristikama i mogućnostima prilagođene prikazu na mobilnim uređajima.

Da biste napravili mobilnu web-stranicu, potrebno je prilikom kreiranja projekta ili dodavanja nove forme izabrati predložak Mobile Web Form. Nakon toga će se u prozoru Toolbox pojaviti nova skupina kontrola pod nazivom Mobile Web Forms.

Mobilne web-forme mogu se prikazivati na cijeloj paleti mobilnih uređaja, od najjednostavnijih mobitela s mogućnošću prikazivanja stranica WAP-a do ručnih računala koja podržavaju gotovo sve funkcionalnosti "pravih" preglednika. Dakako, te se stranice mogu pregledavati i pomoću preglednika na osobnom računalu. Naravno, vi se oko kompatibilnosti ne morate brinuti – kôd koji se šalje klijentskom pregledniku automatski će se prilagođavati njegovu tipu.

Tools), označiti mapu koju želimo promovirati u aplikaciju, ući u prozor Properties i kliknuti na Create, kao što vidimo na slici 10-19. Na taj način možete promovirati i virtualne i fizičke mape.



Slika 10-19:
Promoviranje mape u web-aplikaciju

Komunikacija servera i klijenta

Skripte okupljene pod kapom web-aplikacije imaju određene zajedničke karakteristike, no na kraju se sve uvijek svodi na to da su skripte vrlo samostalne. Web-stranice od svojih početaka funkcioniraju na isti način – korisnik zatraži određenu stranicu, server mu je pošalje i više ne komuniciraju. Među klijentom i serverom ne postoji stalna veza, svaki zahtjev za stranicom zasebna je komunikacija, koja prestaje nakon što se stranica isporuči.

Naravno, poslužitelj i klijent razmjenjivat će i druge podatke osim web-stranice. Većina te komunikacije obavlja se "skriveno", preko zaglavlja web-dokumenta kojeg posjetitelj najčešće nije niti svjestan. Klijent prilikom traženja stranice šalje zaglavlje koje sadrži brojne informacije o tipu klijenta, stranici koju traži, kolačićima i sličnom. Server prilikom odgovaranja na upit također vraća zaglavlje u kojem daje informacije o stranici koju šalje. Zahvaljujući toj međusobnoj komunikaciji moguće je prenošenje vrijednosti među stranicama koje bi inače bile dva potpuno odvojena svijeta.

Prenošenje vrijednosti među web-stranicama je općenito moguće na dva načina – preko spomenutog zaglavlja ili preko parametara u adresi stranice. Vraćanje stranice poslužitelju (*postback*) riješeno je na ovaj prvi način i u potpunosti automatizirano, no kad spajamo različite skripte moramo se pouzdati u vlastite sposobnosti iskorištavanja tih načina komunikacije.

Slanje parametara preko adrese naziva se *get*, a slanje preko zaglavlja stranice naziva se *post*. Adekvatnim parametrom u HTML-u, u formularu je moguće koristiti oba načina.

Treba odmah na početku naglasiti da na ovaj način nije uobičajeno prenositi velike količine podataka. Primjerice, nikad nećete prenositi cijeli zapis baze, nego samo njegovo identifikacijsko polje, a ostale podatke ćete pomoću posebnog upita nanovo izvući iz baze.

Prenošenje parametara ne mora vršiti isključivo između skripti unutar iste web-aplikacije. Parametre možete prebacivati i skriptama druge aplikacije, čak i ako se nalaze na drugom serveru.

Bilo kakvo povezivanje skripata zapravo je obično linkanje. Primjerice, želimo li sa stranice Skripta1.aspx omogućiti odlazak na stranicu Skripta2.aspx, dovoljno je postaviti link (bilo u HTML-u, bilo pomoću web-kontrole HyperLink) na tu stranicu. Primjerice:

```
<asp:HyperLink id="HyperLink1" runat="server" NavigateUrl=" Skripta2.aspx">Skok na
drugu stranicu</asp:HyperLink>
// ili jednostavnije:
<a href="Skripta2.aspx">Skok na drugu stranicu</a>
```

Želimo li u adresi prenijeti neki parametar, koristit ćemo sljedeći izraz:

```
<a href="Skripta2.aspx?parametar1=vrijednost&parametar2=vrijednost">Klikni me</a>
```

U odredišnoj ćemo skripti proslijeđeni parametar pročitati na sljedeći način:

```
string parametar1 = Request.QueryString["parametar1"];
string parametar2 = Request.QueryString["parametar2"];
```

Naravno, u svojoj skripti parametar ne morate pridružiti varijabli – možete ga smjestiti u svojstvo Text neke kontrole te tako ispisati na stranici ili iskoristiti na neki drugi način.

Parametrima proslijeđenima kroz formular pristupamo ovako:

```
string parametar1 = Request.Form["parametar1"];
string parametar2 = Request.Form["parametar2"];
```

Na taj način, iako to u praksi najčešće nije potrebno, možete pristupati i vrijednostima polja generiranih od strane web-kontrola.

Želite li za startnu stranicu koja se otvara u pregledniku nakon kompajliranja postaviti neku drugu umjesto prve, u Solution Exploreru kliknite desnom tipkom miša i iz izbornika izaberite "Set As Start Page".

Osim linkanja koje zahtijeva da posjetitelj klikne na vezu, korisnika možete prebaciti na drugu skriptu naredbom u kodu. Primjerice, da smo željeli u našem primjeru Obrasca za kontakt nakon uspješnog slanja poruke posjetitelja preusmjeriti na posebnu stranicu za zahvalom, koristili bismo sljedeću liniju:

```
Response.Redirect("zahvala.aspx");
```

I u ovom slučaju možemo prosljeđivati parametre u adresi – dovoljno ih je samo dopisati.

Metodu Response.Redirect() radi tako da javi pregledniku (klijentu) neka zatraži drugu stranicu, što ovaj posluša i na taj način preusmjeri korisnika. Želite li da se redirekcija obavi bez znanja klijenta, koristite metodu Server.Transfer().

Sesijske varijable

I sami ste vjerojatno zaključili kako je prenošenje parametara na ovaj način prilično nepraktično. Naime, mogu se prenositi samo parametri tipa *string* (doduše, oni se mogu konvertirati u ostale tipove u kodu, no to može biti nespretno i nesigurno), a i postoji mogućnost da korisnik "upadne" u parametre i promijeni ih.

Stoga u ASP.NET-u (kao i njegovim starijim verzijama) poslužitelj za svakog posjetitelja održava tzv. sesiju. Naime, prvi put kada neki posjetitelj zatraži neku skriptu naše aplikacije, otvara se nova sesija vezana uz njega. Ta sesija će posjetitelja pratiti do kraja njegova posjeta našoj web-aplikaciji.

Određivanje kraja sesije vrlo je neprecizno. Naime, kako ne postoji stalna veza između klijenta i servera, server ne zna kada je posjetitelj otišao s njegovih stranica, a kada samo nešto čeka i planira nastaviti surfanje po njima. Stoga se za kraj sesije najčešće uzima 20 minuta od zadnje aktivnosti na stranicama. Znači, ako korisnik 20 minuta ništa ne dira po vašim stranicama, njegova će sesija biti ugašena. On se možda vrati nakon 25 minuta, no u tom će slučaju za njega biti kreirana nova sesija.

Sesija može pamtiti varijable. Primjerice, ako u kodu neke stranice pridružite vrijednost varijable na sljedeći način:

```
Session["Ime varijable"] = "vrijednost";
```

...ona će biti dostupna i na svim ostalim stranicama koje će nakon ove taj posjetitelj otvoriti. Pazite – vrijednost varijable bit će dostupna samo za tog istog posjetitelja, a ne i za sve ostale posjetitelje. Svaki od njih će imati vlastite sesijske varijable s vlastitim vrijednostima.

Sesijske varijable kojima nije pridružena vrijednost imaju vrijednost *null*.

Sesijske varijable nije potrebno deklarirati niti im je moguće odrediti tip (uvijek su tipa *object*). To u praksi znači da ćete im moći pridruživati vrijednosti bilo kojeg tipa. Primjerice:

```
Session["Varijabla"] = "1";
Session["Varijabla"] = 3;
```

Iako ova praksa izgleda primamljivo, preporučujemo vam da to ne koristite radi preglednosti i dosljednosti – radije napravite dvije različite sesijske varijable, za svaki tip vrijednosti po jednu. Ukoliko vam neka od njih ne treba, napišite sljedeće:

```
Session.Remove("Ime varijable");
```

Međutim, najvažniji razlog je nužnost konverzije prilikom korištenja s ostalim varijablama. Primjerice, prilikom čitanja vrijednosti sesijske varijable i njezina spremanja u varijablu tipa *string*, morat ćemo napraviti konverziju:

```
string varijabla = Session["Ime varijable"].ToString();
```

Sesijske se varijable čuvaju na poslužitelju i posjetitelj im ne može pristupiti niti znati da su kreirane. Može, međutim, na razne načine inicirati prekid sesije, pa stoga ona nije potpuno pouzdan način čuvanja vrijednosti.

Pomoću ovih varijabli najčešće se rješavaju stvari poput postavki posjetitelja koje treba pamtiti tijekom cijelog posjeta ili brojanje stranica koje je posjetio tijekom jednog posjeta.

Vjerojatno ste iz sintakse primijetili da se zapravo radi o kolekciji. Dakle, postoji mogućnost da nad njom radimo sve što i s klasičnom kolekcijom, kao što je šetanje kroz sve njene vrijednosti. Uzmimo za primjer da smo postavili sljedeće serijske varijable:

```
Session["Jedan"] = "1";
Session["Dva"] = 3;
Session["Tri"] = true;
```

Kroz cijelu kolekciju tih varijabli možemo proći na jedan od sljedeća dva načina:

```
for (int n = 0; n < Session.Count; n++)
{
```

```
    Response.Write(Session[n] + "<br>");
}

// ili...

foreach (string x in Session)
{
    Response.Write(Session[x] + "<br>");
}
```

Dakle, varijablama je moguće pristupiti preko ključa (prvi dio primjera) i preko indeksa (drugi dio).

Naredba Response.Write jednostavno ispisuje parametar koji joj damo na web-stranicu. Ona se u principu ne koristi u pozadinskom kodu (posebno ne metodi Page_Load jer ispisuje stvari na sam početak dokumenta, prije zaglavlja HTML-a, što nije pravilno), no može poslužiti za demonstraciju.

Kolačići

Za kolačiće (engl. *cookies*) ste sigurno čuli, ako ni zbog čega drugoga, onda zbog poslovične fobije od gubitka privatnosti. Tehnički, radi se o malenim tekstualnim datotekama na posjetiteljevu računalu u koje web-skripte mogu zapisivati neke njima potrebne podatke.

Sustav za održavanje sesija također koristi kolačić u koji zapisuje identifikacijski kôd sesije.

I kolačići su vezani isključivo uz jednog korisnika, no oni se ne brišu završetkom sesije, već ostaju i za sljedeći posjet, koji može biti za nekoliko sati, nekoliko dana ili čak nekoliko mjeseci.

Drugim riječima, želite li neke vrijednosti vezane uz određenog posjetitelja zapamtiti i nakon kraja sesije, koristit ćete kolačiće.

Kolačiće koristimo nešto drugačije. Prvi red predstavlja pisanje, a drugi čitanje (uočite razlike u naredbama!):

```
Response.Cookies["Ime varijable"].Value = "vrijednost";
string varijabla = Request.Cookies["Ime varijable"].Value;
```

Budući da je vrijednost kolačića tipa string nije bila potrebna konverzija. Međutim, u njega ne možete spremiti, primjerice, broj – ako ga prije ne konvertirate:

```
Response.Cookies["Nekakav broj"].Value = 345.ToString();
```

Međutim, s kolačićima možemo raditi mnoge druge stvari. Osim svojstva Value, svaki kolačić ima neka dodatna svojstva, od kojih ćemo kao najzanimljiviji izdvojiti Expires. Njime definiramo rok trajanja kolačića. Primjerice, sljedeći kolačić će trajati do 27. svibnja 2005. u 10 sati i 50 minuta:

```
Response.Cookies["Ime varijable"].Expires = new DateTime(2005, 5, 27, 10, 50, 00);
```

Rok trajanja možemo odrediti i relativno u odnosu na trenutni datum i vrijeme. Evo kolačića koji će trajati deset dana (od trenutka izvršenja ove naredbe):

```
Response.Cookies["Ime varijable"].Expires = DateTime.Now.AddDays(10);
```

Aplikacijske varijable

I sesijske varijable i kolačići odnose se na samo jednog posjetitelja. Što kada želimo spremiti neku vrijednost koja treba biti dostupna svim korisnicima web-aplikacije, svim posjetiteljima naših web--stranica? U tom ćemo slučaju koristiti aplikacijske varijable.

Sintaksa njihova korištenja ista je kao i kod sesijskih varijabli. Pogledajmo:

```
Application["Ime varijable"] = "vrijednost";
varijabla = Application["Ime varijable"].ToString();
```

Sve napisano za sesijske varijable vrijedi i ovdje, osim činjenice da su aplikacijske dostupne svim korisnicima web-aplikacije. Međutim, iz tog razloga mogu uzrokovati probleme.

Zamislimo situaciju da u nekoj skripti na stranici imamo sljedeći izraz (prvi redak je za slučaj da aplikacijska varijabla nije definirana):

```
if (Application["Broj"] == null) Application["Broj"] = 0;
Application["Broj"] = (int)(Application["Broj"]) + 1;
```

Zadatak drugog retka je da poveća vrijednost navedene aplikacijske varijable za jedan. No što ako u isto vrijeme dva korisnika aplikacije pokrenu istu naredbu? Nastat će problem. Zato kod pridruživanja vrijednosti aplikacijskim varijablama treba koristiti zaključavanje. U sljedećem primjeru ćemo zaključati aplikacijske varijable, napraviti promjenu i zatim otključati zaključano:

```
Application.Lock();
if (Application["Broj"] == null) Application["Broj"] = 0;
Application["Broj"] = (int)(Application["Broj"]) + 1;
Application.UnLock();
```

Zaključavanje odnosno otključavanje uvijek koristite samo i isključivo neposredno prije odnosno poslije pridruživanja vrijednosti kako bi aplikacijske varijable bile zaključane što kraće. Naime, dok su aplikacijske varijable zaključane svi će ostali posjetitelji čekati na njihovo otključavanje, što može drastično utjecati na performanse.

Osim spomenutih načina, ASP.NET omogućava još jedno mjesto pamćenja varijabli na aplikacijskom nivou. Radi se o zajedničkim (statičnim) varijablama klase koje, kao što smo spomenuli u šestom poglavlju, poprimaju samo jednu vrijednost bez obzira na broj instanciranih objekata i zajedničke su svim instancama.

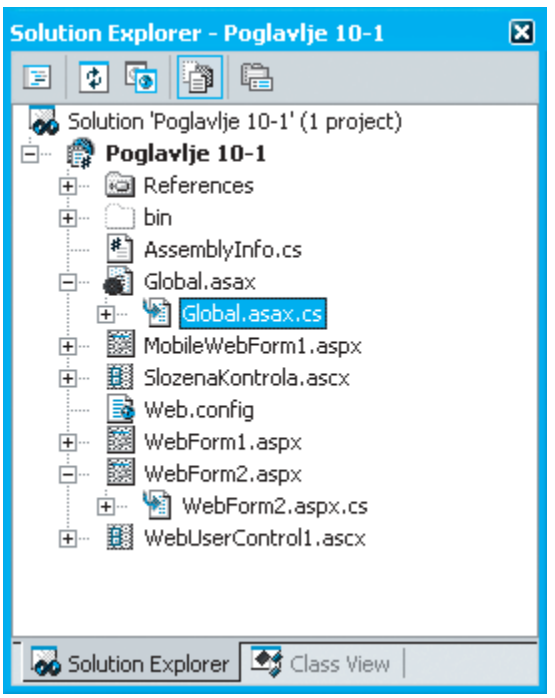
Događaji aplikacije

Kao svaki pravi objekt, i aplikacija ima svoje događaje. Događaje aplikacije pronaći ćete u datoteci Global.asax koja je automatski kreirana prilikom stvaranja projekta.

U njoj su već predefinirane metode povezane na osnovne aplikacijske događaje. Mi ćemo ovdje spomenuti četiri osnovne, one vezane uz događaje početka i završetka aplikacije te početka i završetka sesije.

Najčešća namjena tih metoda je postavljanje inicijalnih vrijednosti aplikacijskih odnosno sesijskih varijabli (tako da izbjegnemo onu provjeru Application[“Broj”] == null iz nedavnog primjera). Evo jednog vrlo čestog primjera – brojanje trenutno aktivnih posjetitelja na stranicama:

```
protected void Application_Start(Object sender, EventArgs e)
{
    Application["Broj posjetitelja"] = 0;
}

protected void Session_Start(Object sender, EventArgs e)
{
    Application.Lock();
    Application["Broj posjetitelja"] = (int)(Application["Broj posjetitelja"]) + 1;
    Application.UnLock();
}

protected void Session_End(Object sender, EventArgs e)
{
    Application.Lock();
    Application["Broj posjetitelja"] = (int)(Application["Broj posjetitelja"]) - 1;
    Application.UnLock();
}
```



Slika 10-20: ***Datoteka Global.asax automatski je kreirana prilikom stvaranja projekta.***

Što ovaj primjer radi? Najprije u metodi vezanoj uz događaj starta aplikacije postavljamo inicijalnu vrijednost aplikacijske varijable Broj posjetitelja. U metodu događaja koji nastupa prilikom otvaranja nove sesije (kako se sesija otvara za svakog novog posjetitelja, događaj nastupa prilikom dolaska posjetitelja na stranice) povećavamo varijablu Broj posjetitelja za jedan, dok prilikom zatvaranja sesije (znači, kad posjetitelj ode sa stranica), tu varijablu smanjujemo za jedan. Na taj način u varijabli Broj posjetitelja uvijek imamo aktualan broj posjetitelja na stranici.

Događaji vezani uz kraj sesije i aplikacije (ovaj potonji ne koristimo u primjeru) nisu pouzdani. Uzmimo banalan primjer – grešku na serveru koja dovede do naglog prekida izvršavanja web-aplikacije. U tom slučaju događaji kraja sesije i aplikacije uopće neće nastupiti. Drugim riječima, u njih ne biste smjeli stavljati stvari koje su ključne za aplikaciju. Primjerice, ne bi bilo pametno napraviti da se nešto broji u aplikacijskog varijabli s namjerom da se prilikom gašenja aplikacije to zapiše u bazu podataka.

Kod zatvaranja sesije vrijedi ista priča, uz nekoliko iznimki. Primjerice, nećete moći pomoću tog događaja posjetitelju ispisati poruku u stilu "Hvala što ste posjetili naše stranice" jer ih je on u tom trenutku već davno napustio, no moći ćete, primjerice, mijenjati aplikacijske varijable kao u primjeru – ako se dogodi havarija spomenuta u prošlom odlomku, ionako će sve vrijednosti aplikacijskih varijabli biti izgubljene pa njihova točnost pada u drugi plan.

Web-aplikacija ima i brojne druge događaje. Primjerice, pomoću događaja BeginRequest i EndRequest možemo se ubaciti na početak odnosno kraj svakog odgovora poslužitelja na upit klijenta dok se aplikacijski događaj Error javlja svaki put kada se dogodi neuhvaćena greška pa ga možete iskoristiti za nekakvo izvještavanje o greškama.

Postavke stranice

Sigurno ste na vrhu stranice u HTML-u primijetili da je prvih par redova žute boje. Te su naredbe drugačije od ostatka kôda i nazivaju se direktive. Njima definiramo određene parametre vezane uz stranicu.

Najkorištenija direktiva je ona nazvana Page, a izgleda otprilike ovako:

```
<%@ Page language="c#" Codebehind="WebForm2.aspx.cs" AutoEventWireup="false"
Inherits="Poglavlje_10_1.WebForm2" %>
```

Kad kreirate novu stranicu u Visual Studiju bit će generirana otprilike takva direktiva. Iz nje trenutno možemo iščitati sljedeće: bit će pisana u jeziku C#, pozadinski kôd nalazi se u datoteci WebForm2.aspx.cs i nasljeđuje klasu Poglavlje_10_1.WebForm2.

Parametar AutoEventWireup nešto je složeniji – sjetimo se da uz svaki događaj trebamo vezati neku metodu koju želimo da se izvrši prilikom tog događaja. Međutim, ako postavimo ovu vrijednost na *true* (ili je ne navedemo – *true* je njezina *defaultna* vrijednost), to povezivanje ne moramo radi-

ti – sustav će sam pozivati metode Page_Init i Page_Load bez obzira na to što nisu povezane sa "svojim" događajima. Visual Studio stavlja ovu vrijednost na *false* stoga što on u automatski generiranom pozadinskom kodu povezuje događaj Load s metodom Page_Load. Stoga ako uključimo još i ovu opciju ta će se metoda pozivati dva puta!

Direktiva Page se može koristiti samo jednom na svakoj stranici. Kontrole ne mogu imati ovu direktivu, kod njih se koristi slična direktiva Control, s nešto manje dostupnih parametara.


```
<%@ Page ... Buffer="false" ... %>
```

Ako ne navedemo parametar Buffer ili ga postavimo na *true*, prvo će poslužitelj generirati kompletnu web-stranicu namijenjenu klijentu, a tek onda je zaista i poslati. Ukoliko stavimo *false*, stranica će se klijentu slati dio po dio, kako koji bude generiran. U praksi uključeni *buffer* omogućava da u zadnji čas otkažete slanje stranice posjetitelju i preusmjerite ga na neku drugu stranicu.

```
<%@ Page ... EnableSessionState="false" ... %>
```

Isključivanjem opcije EnableSessionState isključit ćete stranici mogućnost da koristi sesijske varijable. Ukoliko pak tom parametru pridružite vrijednost ReadOnly, stranica će sesijske varijable moći samo čitati.

```
<%@ Page ... EnableViewState="false" ... %>
```

Parametar EnableViewState uključuje odnosno isključuje pamćenje stanja kontrola na stranici. Imajte na umu da na ovaj način nećete u potpunosti isključiti postojanje skrivenog parametra "__VIEWSTATE" – on će i dalje sadržavati mali izraz pamteći neka osnovna stanja stranice.

```
<%@ Page ... EnableViewStateMac="true" ... %>
```

Uključite li opciju EnableViewStateMac, skriveni parametar stanja bit će kodiran kako bi se spriječila mogućnost korisnikovog mijenjanja tog parametra. Imajte na umu da uključivanje ove opcije utječe na performanse servera pa ga koristite samo na stranicama na kojima je integritet podataka ključan.

```
<%@ Page ... ErrorPage="greske.aspx" ... %>
```

Parametrom ErrorPage određujete stranicu na koju će posjetitelj biti preusmjeren u slučaju nastupa neuhvaćene greške.

```
<%@ Page ... RequestEncoding="iso-8859-2" ResponseEncoding="iso-8859-2" ... %>
```

Parametri RequestEncoding i ResponseEncoding određuju način prikazivanja "neengleskih" znakova na stranicama. Kod .NET-a *defaultna* postavka je UTF-8, univerzalni set znakova koji pokriva većinu svjetskih jezika, što uključuje i naš. Međutim, možete poželjeti da se posjetitelju web-stranice poslužuju na nekoj drugoj kodnoj stranici.

Parametar RequestEncoding govori u kojem setu znakova stranica očekuje zahtjeve od klijenta, dok ResponseEncoding određuje kako će poslužitelj odgovoriti na zahtjev odnosno u kojem setu znakova će stranica biti poslana klijentu.

U navedenom primjeru stranica će očekivati upit i na njega odgovarati koristeći centralnoeuropski ISO-standard.

Tablica 10-2: ***Neki od dostupnih encodinga za ispravan prikaz "naših znakova"***

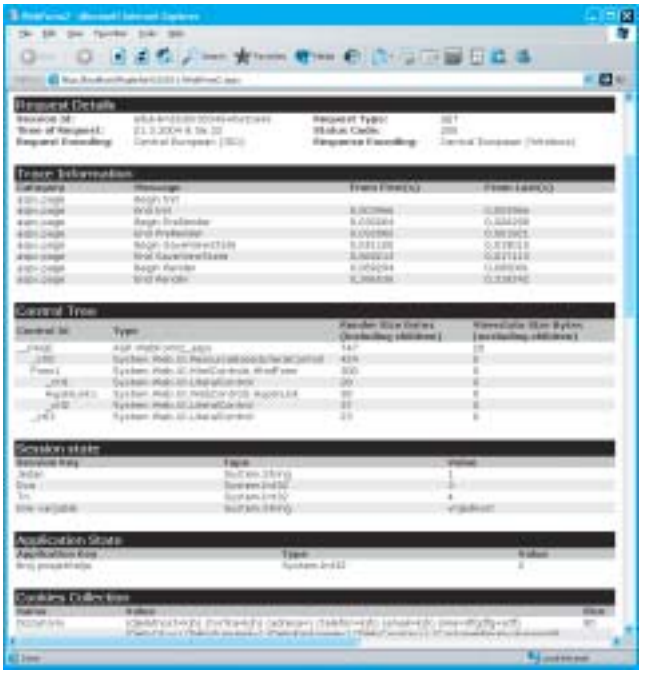
Kratica	Ime
utf-8	Unicode (UTF-8)
iso-8859-2	Central European (ISO)
windows-1250	Central European (Windows)
ibm852	Central European (DOS)

U tablici 10-2 možete vidjeti neke od mogućnosti za prikaz "naših znakova". Radi kompatibilnosti preporučujemo vam da koristite Central European (ISO) – njega podržava najveći broj preglednika na svim sustavima, iako će kroz koju godinu, kada postane još bolje prihvaćen, Unicode biti najbolji izbor.

```
<%@ Page ... Trace="true" ... %>
```

Praćenje parametara stranice koje uključujemo parametrom Trace jedno je od najkorisnijih svojstava ASP.NET-a, pogotovo u slučajevima kad nešto krene po zlu, a vi nikako ne možete otkriti o čemu se radi.

Ono, naime, na kraj stranice dodaje nekoliko tablica popunjenih svim relevantnim podacima o stranici (vidi sliku 10-21). Ti podaci uključuju osnovne informacije (identifikacijsku oznaku sesije, vrijeme, korištene setove znakova, tip zahtjeva, status), informacije o procesiranju stranice, stablo kontrola, popis svih sesijskih i aplikacijskih varijabli zajedno s njihovim tipovima i vrijednostima,



Slika 10-21: ***Uključeno praćenje parametara stranice donos pregršt informacija.***

U praćenje parametara stranice možete dodavati i svoje parametre iz pozadinskog kôda. To može poslužiti za ispisivanje vrijednosti varijabli za koje niste sigurni kako se ponašaju. Sintaksa je sljedeća:

```
Trace.Write("Varijabla Var1 je jednaka " + Var1.ToString());
```

sve dostupne kolačiće, detaljne informacije o zaglavlju poslane od strane posjetiteljeva preglednika (iz kojih, primjerice, možemo vidjeti koji preglednik posjetitelj koristi) kao i kompletan popis serverskih varijabli koje sadrže apsolutno sve informacije o poslužitelju, klijentu i njihovoj vezi.

I parametara i direktiva ima još. Jednu od direktiva, Register, već smo upoznali prilikom korištenja korisničkih kontrola na stranicama, a još jednu među njima upoznat ćemo kasnije u poglavlju, u priči o *cacheiranju*. Ostale koje se rjeđe koriste ostavljamo vam na samostalno proučavanje – putanja u dokumentaciji je .NET Development > .NET Framework SDK > .NET Framework > Reference > ASP.NET Syntax > Web Forms Syntax > Directive Syntax.

Konfiguracijske datoteke

Određene postavke možete postaviti i na nivou aplikacije smještajući ih u konfiguracijske datoteke. Te će se postavke poštovati na svakoj stranici aplikacije. Na taj način može se definirati neopisivo velik broj stvari, a mi ćemo, kao i uvijek, obraditi korištenije i zanimljivije.

Ipak, prije nego što krenemo na seciranje konfiguracijskih datoteka, red je da objasnimo njihovu hijerarhiju. Postoji, naime, jedna glavna konfiguracijska datoteka koja se zove "machine.config" i nalazi se u mapi "C:\WINDOWS\Microsoft.NET\Framework\v1.1.4322\CONFIG". Ona predstavlja konfiguracijsku datoteku za cijelo računalo – sve što je u njoj definirano vrijedit će u svim web-aplikacijama na tom računalu. Osim ako neke druge konfiguracijske datoteke ne prepišu određenu opciju.

Naime, svaki web-*site* također može imati svoju konfiguracijsku datoteku. Ona se međutim naziva "web.config" i nalazi se u njegovoj osnovnoj mapi, tzv. *rootu*. Postavke u toj datoteci prepisat će one u glavnoj konfiguracijskoj datoteci.

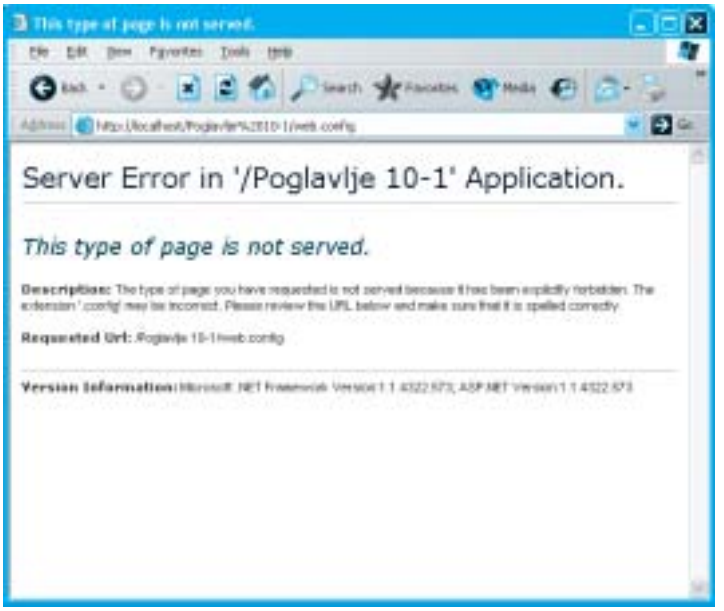
Nadalje, svaka web-aplikacija na određenom *siteu* može imati svoju konfiguracijsku datoteku. Primjerice, aplikacija s adresom "http://localhost/WebApplication1/" u svojoj će mapi (WebApplication1) također vjerojatno imati svoju konfiguracijsku datoteku nazvanu "web.config".

Kada naša web-aplikacija traži koje postavke treba koristiti, prvo zaviruje u glavnu konfiguracijsku datoteku i uzima tamo postavljene opcije. Zatim gleda datoteku koja se nalazi na adresi "http://localhost/web.config" (uvjetno rečeno, jer je tim datotekama nemoguće pristupiti na taj način) i opcijama koje tamo pronađe prepiše one koje je pokupila iz glavne datoteke. Konačno, ulazi u konfiguracijsku datoteku same aplikacije (dakle, uvjetno rečeno, "http://localhost/WebApplication1/web.config") i tamo pronađenim vrijednostima postavlja konačnu konfiguraciju.

Zvuči jednostavno? Nismo još gotovi. Naime, konfiguracijska datoteka višeg stupnja može zabraniti prepisivanje parametara pomoću konfiguracijske datoteke nižeg stupnja. Drugim riječima, ako u datoteci "machine.config" odredite da se ne može prepisivati opcije pomoću datoteka "web.config", onda će svaki pokušaj prepisivanja rezultirati greškom.

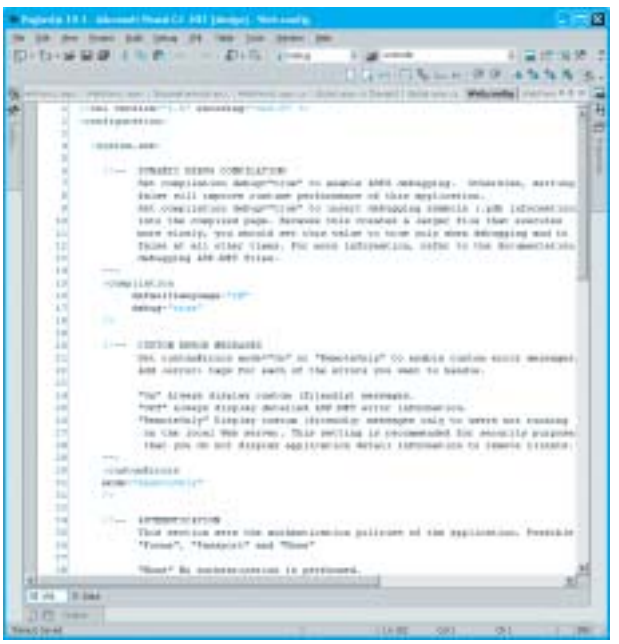
Preporučujemo vam da konfiguracijske datoteke, kao uostalom i sve datoteke koje koristite u programiranju, otvarate i mijenjate pomoću Visual Studija. Naime, sve se datoteke prema inicijalnim postavkama spremaju u formatu Unicode, što mnogi editori ne podržavaju. Nakon snimanja datoteka s takvim programima one bi mogle ispasti neispravne i neupotrebljive.

Cijela ova priča nije od krucijalne važnosti – u pravilu ćete se baviti isključivo opcijama na nivou konfiguracijske datoteke za aplikaciju. Što postavite tamo, vrijedit će za cijelu vašu aplikaciju.



Slika 10-22: ***Poruka koja se javi kad pokušate preko weba pristupiti datoteci web.config***

Kako prema inicijalnim postavkama prepisivanje nije zabranjeno, moći ćete definirati sve što želite. Ipak, naletite li na neki strogo konfiguriran poslužitelj, ove će vam informacije uštedjeti mnogo truda.



Slika 10-23: ***Datoteka web.config otvorena u Visual Studiju***

Prilikom svake promjene konfiguracijske datoteke aplikacija će se automatski restartati i nove će vrijednosti početi vrijediti.

Igranje s opcijama konfiguracijske datoteke "web.config" vrlo je jednostavno jer se te datoteke automatski kreiraju prilikom stvaranja novog projekta, a neke su mogućnosti i objašnjene u samoj datoteci u obliku komentara.

Konfiguracijske datoteke osjetljive su na razliku između malih i velikih slova.

Konfiguracijske datoteke zapravo su XML-dokumenti. Iako njih još zvanično nismo obradili (to nas čeka u sljedećem poglavlju), vjerujemo da nećete imati problema sa savladavanjem njihove sintakse.

Statične varijable

U konfiguracijsku datoteku možemo zapisati neke statične varijable koje će vrijediti za cijelu aplikaciju. To se najčešće koristi za spremanje *connection stringa* za pristup bazi podataka, kao što ćemo vidjeti u primjeru nešto kasnije u poglavlju. Evo primjera:

```
<configuration>
   <appSettings>
      <add key="ConnectionString" value="neki connection string"/>
   </appSettings>
</configuration>
```

Sve se postavke u konfiguracijskoj datoteci nalaze između *tagova* <configuration> i </configuration>. Između njih stavljamo *tagove* <appSettings> i </appSettings> unutar kojih, pomoću vidljive sintakse, dodajemo ime varijable (parametar *key*) i vrijednost (parametar *value*).

Toj vrijednosti iz pozadinskog kôda možemo pristupiti na sljedeći način (uočite nužno navođenje *namespacea* na vrhu):

```
using System.Configuration;
...
string cs = ConfigurationSettings.AppSettings["ConnectionString"];
```

Postavke stranice

Neke postavke koje smo definirali na stranici možemo, umjesto da ih postavljamo na svakoj stranici posebno, definirati i na nivou cijele aplikacije. Naravno, ukoliko stranica ima drugačije postavljenu vrijednost, ova u konfiguracijskoj datoteci bit će prepisana (ignorirana).

Kodne stranice ćete gotovo sigurno poželjeti definirati na nivou cijele aplikacije jer nema smisla da svaka stranica biva enkodirana na drugi način.

```
<configuration>
   <globalization
      requestEncoding="iso-8859-2"
      responseEncoding="iso-8859-2"
   />
</configuration>
```

Neki se drugi parametri koje smo upoznali kod direktive Page u konfiguracijskog datoteci definiraju na ovaj način:

```
<configuration>
   <system.web>
      <pages
         buffer="true"
         enableSessionState="true"
         enableViewState="true"
         autoEventWireup="true"
      />
   </system.web>
</configuration>
```

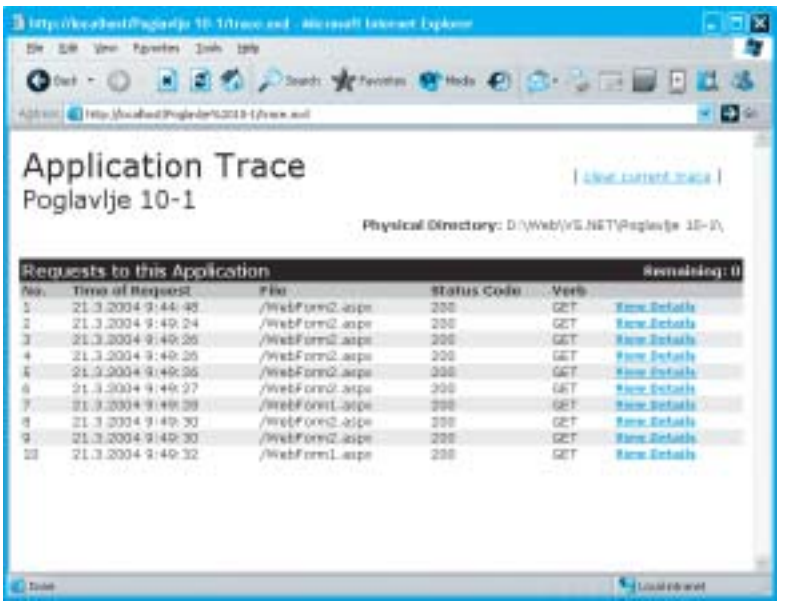
Pažljivo uočavajte gdje su pojedine opcije smještene. Primjerice, opcija <pages> obavezno mora biti u sekciji <system.web>.

Praćenje parametra

Praćenje parametara na nekoj stranici možemo uključiti pomoću direktive Page. Međutim, želimo li uključiti to praćenje za sve stranice unutar aplikacije, uključit ćemo opcije enabled i pageOutput. Prva će općenito uključiti praćenje parametara, dok će druga naložiti da se informacija ispisuje na svakoj stranici.

```
<configuration>
   <system.web>
      <trace
         enabled="true"
         localOnly="true"
         pageOutput="true"
         requestLimit="15"
      />
   </system.web>
</configuration>
```

Ukoliko parametar pageOutput isključimo, do informacija o praćenju ćemo moći doći preko posebne, virtualne adrese – http://localhost/trace.axd. Datoteka na kraju adrese, dakako, ne postoji, no web-poslužitelj će nam vratiti stranicu s popisom svih zahtjeva nad aplikacijom (vidi sliku 10-24). Klikom na link "View Details" možemo vidjeti detalje o svakom zahtjevu kao što je bilo prikazivano na kraju stranice.



Slika 10-24: ***Stranica s popisom svih zahtjeva nad aplikacijom***

Kako zahtjeva u aplikaciji može biti izuzetno puno, parametrom requestLimit određujemo koliko će zadnjih zahtjeva biti pamćeno i prikazano na stranici.

Informacije o praćenju parametara inicijalno su dostupne samo s lokalnog računala. O tome se brine parametar localOnly. Kada bismo ga isključili, te bi informacije mogli vidjeti svi posjetitelji naših stranica.

Upravljanje sesijom

Iako je sesija kod ASP.NET-a u praksi ostala gotovo ista u odnosu na starije inačice, u samoj biti promijenjeno je mnogo toga. Tako, primjerice, za cijeli posao oko održavanja sesija možemo zadužiti neki vanjski server, što posebno dolazi do izražaja u slučajevima kada se više servera brine o istoj web-aplikaciji (što se naziva web-farma). Mi u detalje toga nećemo ulaziti, no zato ćemo spomenuti mogućnost održavanja sesije bez kolačića.

Naime, kao što smo već spomenuli, sustav za održavanje sesije koristi kolačić kako bi zapisao identifikacijski kôd sesije. Ukoliko posjetiteljev preglednik ima isključene kolačiće (da, ima i takvih...), rješenje može biti uključivanje parametra *cookieless* i korištenje tzv. sesije bez kolačića.

```
<configuration>
   <system.web>
      <sessionState
         cookieless="false"
         timeout="20"
      />
   </system.web>
</configuration>
```

U tom će se slučaju identifikacijski kôd prenositi u adresi stranice koja će izgledati, primjerice, ovako:

```
http://localhost/ImeAplikacije/(so5os0550bxammv5a5skvivz)/WebForm1.aspx
```

Korisnik identifikacijski kôd na ovaj način može vrlo jednostavno promijeniti i tako "iskočiti" iz postojeće sesije i napraviti novu. Doduše, ista se stvar može napraviti i s kôdom zapisanim u kolačić, no to je znatno složenije.

Konačno, u konfiguracijskoj datoteci možemo i izmijeniti rok trajanja sesije nakon zadnjeg zahtjeva za stranicom. Parametru *timeout* navodi se vrijeme u minutama.

Poruke o greškama

Iz poruka o greškama može se puno saznati o strukturi i načinu rada web-aplikacije. Stoga je poželjno da takve poruke ne dođu u ruke osobama zlih namjera, a kako su web-stranice dostupne svima, znači da bi svima trebalo prikazivati tek informaciju da je greška nastupila, a detalje zadržati u tajnosti.

```
<configuration>
   <system.web>
      <customErrors
         mode="RemoteOnly"
      />
   </system.web>
</configuration>
```

Ovako postavljena vrijednost parametra određuje da se općenita informacija o grešci (tzv. "prijateljska" poruka odnosno engl. *friendly error message*) pokazuje svim vanjskim posjetiteljima, a korisniku na lokalnom računalu prikazat će se detaljna poruka o grešci.

Parametar mode može poprimiti vrijednosti i "On" (svima će biti prikazana općenita poruka) ili "Off" (svima će biti prikazana detaljna poruka).

Osim toga, možemo definirati da se umjesto generičke osnovne poruke prikaže neka naša. Primjerice, želimo li da svim vanjskim posjetiteljima umjesto osnovne poruke bude prikazana stranica "error.aspx", u konfiguracijsku datoteku ćemo smjestiti sljedeći izraz:

```
<configuration>
   <system.web>
      <customErrors
         mode="RemoteOnly"
         defaultRedirect="error.aspx"
      />
   </system.web>
</configuration>
```

Autentikacija i autorizacija korisnika

Zabrana pristupa nepoznatim (anonimnim) korisnicima često se koristi kod stranica koje nisu namijenjene javnosti. U ASP.NET-u je napravljena izuzetna podloga za implementaciju autentikacije i autorizacije korisnika.

Prema inicijalnim postavkama svatko može pristupiti web-stranicama. Takvi se posjetitelji nazivaju anonimni korisnici.

Autentikacija je proces utvrđivanja tko je došao na stranice, a autorizacija je utvrđivanje ima li on pravo pristupa određenim stranicama.

```
<configuration>
   <system.web>
      <authorization>
         <allow users="*" />
      </authorization>
   </system.web>
</configuration>
```

Ovakva postavka omogućava svim korisnicima da pristupaju našim stranicama.

Kada bismo napisali sljedeće, svima bismo onemogućili pristup:

```
<deny users="*" />
```

Međutim, mi želimo postići da se pristup zabrani samo anonimnim korisnicima, pa ćemo umjesto gornjih izraza napisati ovo:

```
<deny users="?" />
```

Ako posjetitelj dođe na web-aplikaciju s ovakvom postavkom, trebat će se autenticirati. Nakon uspješne autentikacije on neće više biti anoniman korisnik, pa će moći pristupiti aplikaciji.

Autentikaciju možemo napraviti na četiri načina, a to definiramo sljedećim izrazom:

```
<authentication mode="Windows" />
```

Slika 10-25: ***Prozor za upis korisničkog imena i lozinke prilikom korištenja autentikacijskog načina Windows***

Ako je autentikacijski način postavljen na "Windows", znači da će posjetitelji morati upisati korisničko ime i lozinku koji postoje u bazi korisnika poslužitelja (ili domene u kojoj se on nalazi).

Osim autentikacijskog načina Windows (koji je najjednostavniji za implementaciju), dostupni su načini Forms (autentikacija se obavlja putem posebno napravljene stranice s obrascem za unos imena i lozinke), Passport (autentikacija putem Microsoftova servisa .NET Passport) i None (za *siteove* kojima autentikacija nije potrebna ili imaju vlastiti sustav autentikacije).

Isprobavanje autentikacije nećete jednostavno moći obaviti na lokalnom računalu jer vas poslužitelj neće tretirati kao anonimnog korisnika, već kao korisnika s korisničkim računom kojim ste prijavljeni na računalo.

Cacheiranje

Web-stranice često istovremeno posjećuje velik broj posjetitelja. Generiranje stranica za svakoga posebno zahtjevan je posao i postoji mogućnost da vaš server to ne može podnijeti. Stoga u ASP.NET-u postoje tri vrste *cacheiranja* – funkcionalnosti koja omogućava spremanje stranice, njezinih dijelova ili podataka u memoriju. Ako neki posjetitelj zatraži stranicu koja je spremljena u *cache*, ona neće biti ponovo generirana, već će se posjetitelju poslati direktno iz *cachea*.

Imajte na umu da *cacheiranje* nije idealno rješenje u svim prigodama. Općenito možemo reći da ukoliko web-stranica ima puno posjetitelja kojima se prikazuje na isti način *cacheiranje* može biti vrlo korisno i može značajno ubrzati posluživanje stranica. Međutim, u slučaju, recimo, pretraživača u koji svaki posjetitelj upisuje drugačiji izraz, *cacheiranje* nema nikakvog smisla – stranica će se ionako svaki put generirati iznova zbog drugačijih parametara, a stranice spremljene u *cache* nitko neće koristiti pa će nepotrebno zauzimati memoriju i tako usporavati rad poslužitelja.

Cacheiranje stranica

Želimo li neku stranicu uključiti u proces *cacheiranja* moramo joj na početak dodati sljedeću direktivu:

```
<%@ OutputCache Duration="100" VaryByParam="none" %>
```

Direktivi OutputCache u primjeru navodimo dva parametra. Prvi određuje trajanje *cache* u sekundama – u našem slučaju znači da će stranica biti spremljena u memoriji sljedećih sto sekundi.

Ako neki posjetitelj pozove stranicu nakon isteka tog vremena, ona će biti iznova generirana i opet spremljena u *cache* na sljedećih sto sekundi.

Ukoliko će se stranica pozivati s određenim parametrima (bilo u adresi – GET, bilo kroz zaglavlje – POST), poželjet ćete se poigrati parametrom VaryByParam. Naime, pomoću njega možete odrediti hoće li se stranica s drugačijim parametrima tretirati kao sasvim nova stranica ili će, unatoč drugačijim parametrima, posjetitelju biti poslužena ona iz memorije. Ako stavimo vrijednost "none", parametri će biti ignorirani, a ako stavimo zvjezdicu ("*"), parametri će se uvažavati. Moguće je i rješenje između – možete nabrojati parametre koji će biti uvažavani, dok će ostali biti ignorirani. Primjerice:

```
<%@ OutputCache Duration="100" VaryByParam="location;count" %>
```

Osim po parametru, stranicu je moguće diferencirati i prema zaglavlju (VaryByHeader) i nekom vlastitom uvjetu (VaryByCustom).

Ako koristite *cacheriranje* na stranicama koje se generiraju iz baze podatakа, imajte na umu da se promjene na bazi neće manifestirati na stranicama sve dok cache ne istekne i stranica se ponovo generira. Zato je praksa da se kao trajanje *cacheiranja* stavljaju manje vrijednosti.

Cacheiranje dijelova stranice

Na gotovo isti način možete *cacheirati* i dijelove stranice. Doduše, dijelove koje želite *cacheirati* najprije morate pretvoriti u korisničke kontrole te onda kontrolama dodati direktivu OutputCache na isti način kao što smo to u prošlim primjerima dodavali stranicama.

```
<%@ OutputCache Duration="100" VaryByParam="none" %>
```

Kada koristimo direktivu OutputCache na kontrolama, osim već spomenutih parametara, otvara nam se još jedan koji bi trebalo spomenuti – Shared.

Naime, kako kontrole možemo koristiti na više stranica, treba odlučiti smije li poslužitelj istu kontrole na različitim stranicama tretirati kao jednake. Inicijalno će se u memoriju spremati zasebna kopija kontrole za svaku stranicu, a ukoliko to želimo izbjeći, napisat ćemo sljedeće:

```
<%@ OutputCache Duration="100" VaryByParam="*" Shared="True" %>
```

Cacheiranje podataka

Varijable *cachea* su svojevrsna alternativa aplikacijskim varijablama. Naime, jednom postavljena aplikacijska varijabla zauzima prostor u memoriji do kraja izvršavanja aplikacije koja na stabilnim produkcijskim web-poslužiteljima može trajati godinama. Varijable u kodu se čiste iz memorije nakon što je stranica poslužena, sesijske se brišu po isteku sesije, dok aplikacijske traju, traju i traju...

Varijable *cachea* po svemu su identične aplikacijskima uz, naravno, dvije sitnice. Prvo, prilikom mijenjanja njihovih vrijednosti one se automatski zaključavaju i, drugo, poslužitelj se brine da one koje se duže vremena ne koriste automatski izbrišu iz memorije.

Njih se najčešće koristi za, primjerice, spremanje stvari kao što su lista opcija za padajući izbornik (da se ne vuče svaki put iz baze podataka) ili pak neke druge vrijednosti koje su iste za sve korisnike. Naravno, svaki put treba provjeriti je li tako spremljen podatak istekao, no na taj način možemo značajno uštedjeti na resursima poslužitelja.

Varijable *cachea* se koriste prema već viđenom modelu i sve rečeno za sesijske i aplikacijske vrijedi i ovdje:

```
Cache["Ime varijable"] = "vrijednost";
string varijabla = Cache["Ime varijable"].ToString();
```

Međutim, vjerojatnije ćete u ove varijable spremati neke veće količine podataka. Recimo da želimo polje iz primjera Obrazac za kontakt spremiti u *cache* kako se ne bi trebalo iznova puniti prilikom svakog učitavanja stranice. Evo kako ćemo to napraviti:

```
if (Cache["drzaveArray"] == null)
{
   drzave = new string[] { "Hrvatska", "Italija", "Portugal", "Brazil" };
   Cache.Insert("drzaveArray", drzave, null, DateTime.Now.AddMinutes(10),
     TimeSpan.Zero);
}
else
{
   drzave = (string[])(Cache["drzaveArray"]);
}
```

Najprije uvjetom provjeravamo postoji li zapisana vrijednost u varijablu drzaveArray. Ako je jednaka vrijednosti *null* znači da ne postoji, bilo da nikada nije niti postojala, bilo da je istekla. U tom slučaju punimo polje podacima (u našem slučaju na ovaj loš način, kod vas vjerojatno iz neke baze ili datoteke) i zatim ga metodom Insert spremamo u varijablu *cachea*. Prvo navodimo ime varijable, a zatim i vrijednost koju želimo smjestiti u varijablu *cachea*. Sljedeći parametar predstavlja neku zavisnu vrijednost koja će odlučiti o isteku valjanosti *cachea* (mi smo stavili *null,* što označava da

nema zavisnih vrijednosti). Slijedi definiranje roka valjanosti – prvi parametar je određuje konkretnim vremenom (u našem slučaju na vrijeme deset minuta nakon postavljanja), dok drugi određuje rok isteka u relativnom obliku, od zadnjeg pristupa varijabli. Primjerice, da smo tu stavili vrijednost TimeSpan.FromMinutes(1), varijabla bi istekla minutu nakon što je zadnji put korištena (ne kreirana!).

Parametre za istek varijable nije potrebno navoditi, no ako ih navodite, samo jedan od njih smije imati konkretnu vrijednost. Drugim riječima, ako postavite prvi na neku vrijednost (primjerice DateTime.Now.AddMinutes(10)), drugi mora biti TimeSpan.Zero. Ukoliko pak drugi ima neku vrijednost (primjerice TimeSpan.FromMinutes(1)), prvi morate postaviti na DateTime.MaxValue.

Rad s bazama podataka

U pozadini web-stranica najčešće možemo naći baze podataka. Stoga ćemo se sada posvetiti obradi kontrola koje nam u tome mogu puno pomoći – DataGrid, DataList i Repeater. Prvu ćemo nešto detaljnije upoznati, a druge dvije ćemo samo ukratko objasniti. Međutim, kako sve tri rade na sličan princip, i njih ćete brzo "prokužiti".

Naravno, trebat će nam i znanja koja smo usvojili u prošlom poglavlju u kojem smo baze podataka upoznali teoretski i korištenjem u prozorskoj aplikaciji, a ponovit ćemo i neke stvari koje smo spomenuli u prvom dijelu ovog poglavlja.

No krenimo redom...

Kontrola DataGrid

Kontrola DataGrid služi za prikaz podataka iz baze u obliku tablice. Evo krajnje jednostavnog primjera – postavite na stranicu kontrolu DataGrid (imena DataGrid1), a u pozadinski kôd napišite sljedeće:

```
private void Page_Load(object sender, System.EventArgs e)
{
    if (!Page.IsPostBack)
    {
        Bindanje();
    }
}

private void Bindanje()
{
    SqlConnection sqlConn = new
```

```
        SqlConnection(ConfigurationSettings.AppSettings["connstr"]);

    SqlDataAdapter sqlComm = new SqlDataAdapter("SELECT * FROM Orders", sqlConn);

    DataSet ds = new DataSet();
    sqlComm.Fill(ds);

    DataGrid1.DataSource = ds;
    DataGrid1.DataBind();
}
```

Nemojte zaboraviti referencu na *namespaceove* na početku pozadinskog kôda:

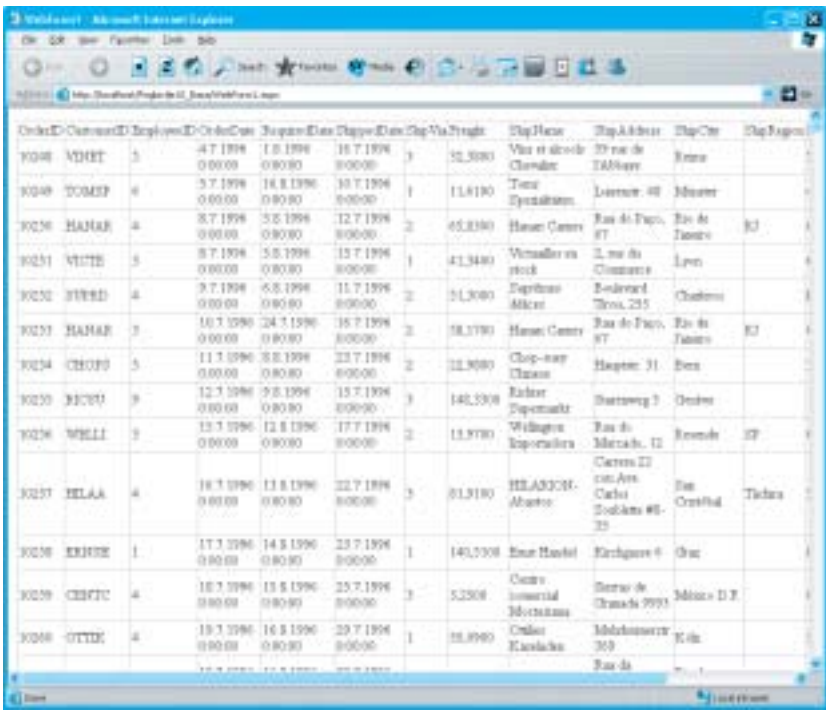
```
using System.Data.SqlClient;
using System.Configuration;
```

Mi ćemo u primjerima koristiti bazu podataka Northwind smještenu na lokalnom SQL Serveru. Naravno, uz sitne modifikacije prema uputama iz prošlog poglavlja, sve će raditi i s nekim drugim izvorom podataka.

Primjećujete da *connection string* izvlačimo iz konfiguracijske datoteke web.config (što je preporučljiva praksa), pa stoga u nju treba dodati sljedeće:

```
<appSettings>
    <add
        key="connstr"
        value="server=(local);database=Northwind;Integrated Security=true;"
    />
</appSettings>
```

Kod povezivanja iz web-aplikacije na SQL Server trebate imati na umu da stranice njemu pristupaju pod posebnim korisničkim računom nazvanim ASPNET. Da bi taj korisnički račun mogao pristupati bazi podataka, potrebno ga je dodati među korisnike SQL Servera i pridružiti mu potrebna prava nad bazom podataka koja će se koristiti.



Slika 10-26: ***Web-aplikacija koja pomoću DataGrida prikazuje tablicu Orders***

Pokrenemo li aplikaciju, dobit ćemo tabelarni prikaz svih zapisa koje smo dohvatili SQL-upitom u funkciji Bindanje(), kao što je prikazano na slici 10-26.

Vizualno dotjerivanje

Osim izuzetno jednostavnog postizanja ove funkcionalnosti, primijetit ćete da je dobivena tablica nevjerojatno ružna. Stoga je treba vizualno dotjerati pomoću brojnih dostupnih svojstava.

DataGrid možete urediti na dva načina – prvi je korištenjem predefiniranih stilova koji se kriju iza linka Auto Format (vidi dno slike 10-27), dok je drugi način ručno podešavanje svojstava u grupama Appearance, Layout i Style. Mi nećemo ulaziti u detalje opisujući karakteristike svakog pojedinog svojstva – vjerujemo da su im imena dovoljno intuitivna.

Zanimljivo je pogledati kako su spomenute stavke manifestirane u kodu. Dosad smo uvijek sva svojstva dodavali kao atribute, no ovdje se situacija nešto razlikuje – neka su svojstva i dalje atributi osnovne kontrole, dok su stilovi pojedinih redova zasebni *tagovi* između otvorenog i zatvorenog *taga* kontrole. Evo primjera:

```
<asp:DataGrid id="DataGrid1" runat="server" BorderColor="#E7E7FF" BorderStyle="None"
BorderWidth="1px" BackColor="White" CellPadding="3" GridLines="None">
   <ItemStyle ForeColor="#4A3C8C" BackColor="#E7E7FF"></ItemStyle>
   <AlternatingItemStyle BackColor="#F7F7F7"></AlternatingItemStyle>
</asp:DataGrid>
```

Slika 10-27:
Svojstva za vizualnu prilagodbu kontrole DataGrid

Prikaz samo određenih polja

Rijetko ćemo na stranici željeti pokazati sva polja koja smo upitom dohvatili. Ponekad ćemo moći jednostavno suziti upit nad bazom (što se preporučuje kada je ikako moguće), no ponekad ćemo "filtriranje" polja trebati napraviti na nivou kontrole. Da bismo to postigli, trebamo napraviti dvije stvari: parametru AutoGenerateColumns postaviti vrijednost na *false* te dodati sekciju Columns

unutar koje navodimo polja (stupce) koja želimo prikazati. Parametrom DataField određujemo ime polja, dok HeaderText definira tekst u zaglavlju.

```
<asp:DataGrid runat="server" AutoGenerateColumns="False">
   <Columns>
      <asp:BoundColumn DataField="OrderID"
         HeaderText="Broj narudžbe"></asp:BoundColumn>
      <asp:BoundColumn DataField="ShipName"
         HeaderText="Ime dostave"></asp:BoundColumn>
      <asp:BoundColumn DataField="ShipAddress"
         HeaderText="Adresa dostave"></asp:BoundColumn>
      <asp:BoundColumn DataField="ShipCity"
         HeaderText="Grad dostave"></asp:BoundColumn>
      <asp:BoundColumn DataField="OrderDate" HeaderText="Datum narudžbe"
         DataFormatString="{0:dd.MM.yyyy.}"></asp:BoundColumn>
      <asp:BoundColumn DataField="Freight" HeaderText="Težina"
         DataFormatString="{0:#.##} kg"
         ItemStyle-HorizontalAlign="Right"></asp:BoundColumn>
   </Columns>
</asp:datagrid>
```

Prilikom formatiranja datuma i vremena pazite – mm je oznaka za minute, a MM za mjesece.

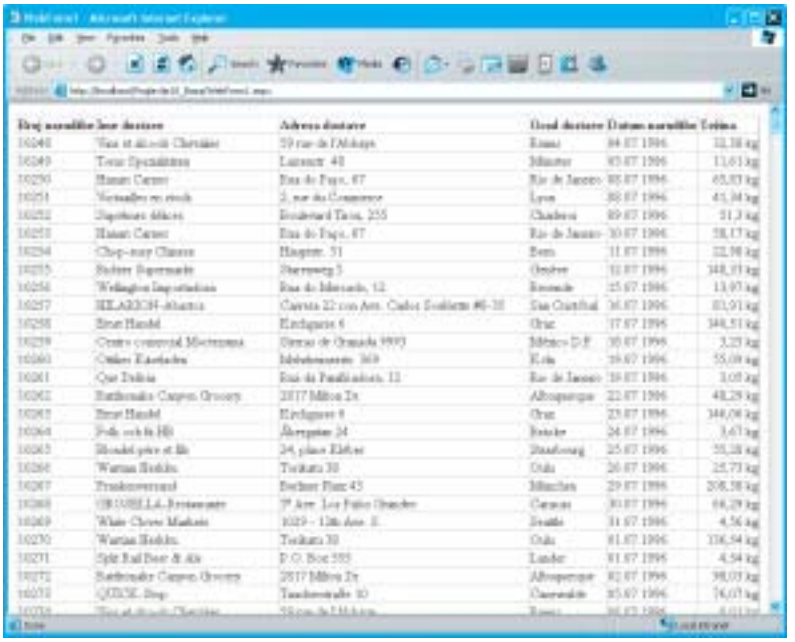
U zadnjim smo stupcima koristili i dva nova svojstva. DataFormatString određuje format na koji će podatak biti prikazan, dok ItemStyle-HorizontalAlign definira vodoravnu poravnatost podataka u stupcu.

Izbor zapisa

Sada ćemo u postojeću tablicu dodati još jedan stupac, no on neće sadržavati vrijednost iz baze podataka, već link. Stoga unutar bloka <columns> dodajte sljedeći redak:

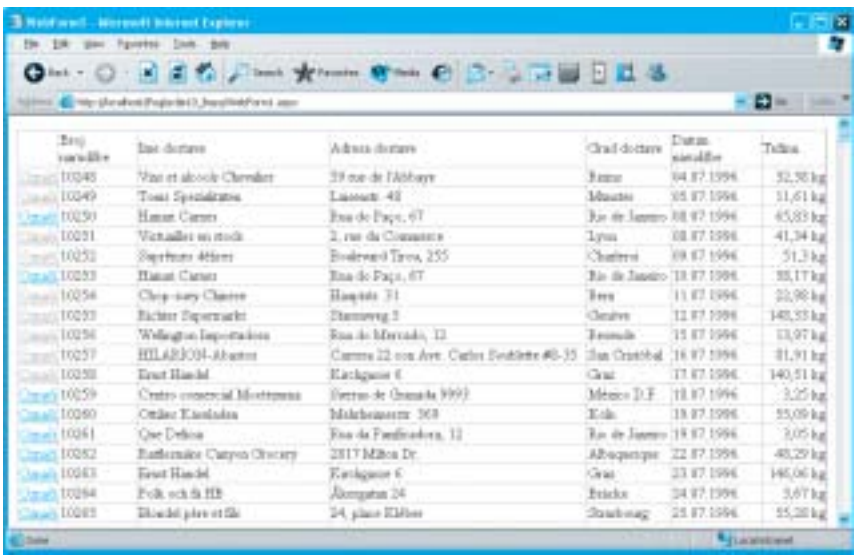
```
<asp:ButtonColumn ButtonType="LinkButton" Text="Označi"
CommandName="oznacavanje"></asp:ButtonColumn>
```

Kada posjetitelj klikne na spomenuti link, nastupit će događaj ItemCommand. Dakle, prebacit ćemo se u dizajnerski način, označiti kontrolu DataGrid i u prozoru Properties dvokliknuti na događaj ItemCommand. Otvorit će nam se metoda u kojoj trebamo dodati željenu funkcionalnost.



Slika 10-28: ***Prikaz samo nekih polja iz baze***

Vrlo je bitno da se prilikom vraćanja stranica serveru (*postbacka*) ne izvršava metoda Bindanje(). Naime, radnje u njoj bi izbrisale događaj ItemCommand i vezana metoda ne bi bila izvršena.



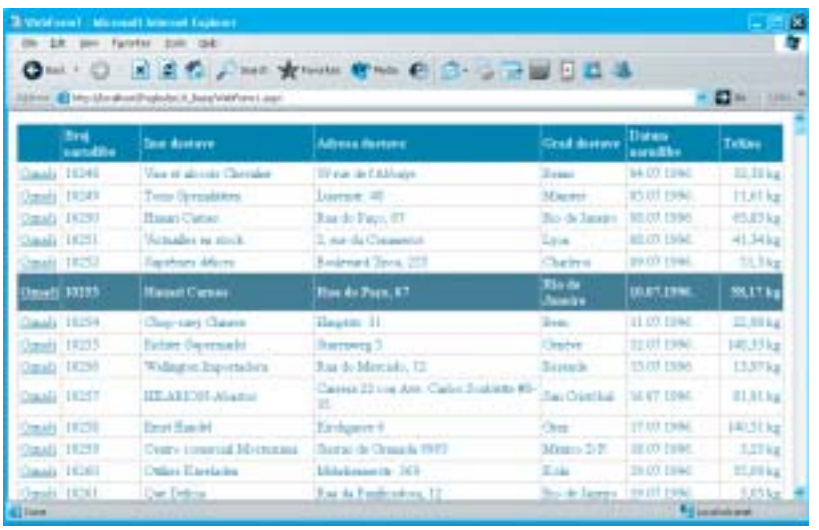
Slika 10-29: ***Novi stupac s linkom***

Događaj ItemCommand nastupa prilikom bilo kakve aktivnosti unutar DataGrida. Među ostalima, to je i klik na jedan od linkova u našem stupcu Označi. U našem primjeru nema drugih događaja, no da ima, mogli bismo ih razlikovati prema parametru CommandName koji smo naveli u *tagu* <asp:ButtonColumn>. Koja je "komanda" nastupila, u funkciji provjeravamo ovako:

```
private void DataGrid1_ItemCommand(object source,
System.Web.UI.WebControls.DataGridCommandEventArgs e)
{
    if (e.CommandName == "oznacavanje")
    {
        DataGrid1.SelectedIndex = e.Item.ItemIndex;
    }
}
```

Kroz parametar e možemo otkriti u kojem se retku nalazio link koji je pritisnut. U gornjem primjeru tu vrijednost pridružujemo svojstvu DataGrid1.SelectedIndex, što će rezultirati označavanjem tog retka.

Kako smo, radi jednostavnosti primjera, isključili stilove, označeni redak vjerojatno nećete uočiti. Da biste ga mogli uočiti, treba definirati neko svojstvo unutar stila SelectedItemStyle.



Slika 10-30: ***Klikom na link "Označi" redak se – označio.***

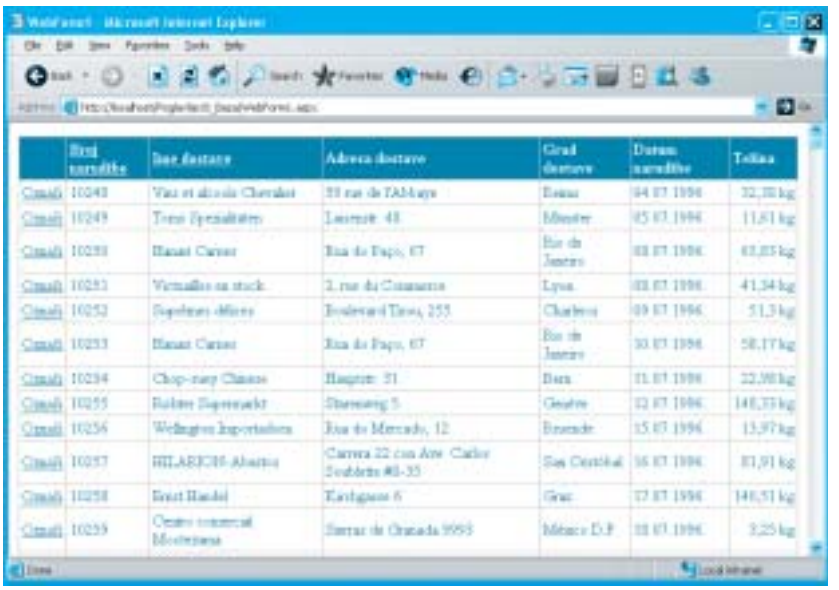
Ovo je najjednostavnije što smo mogli učiniti s tim poljem. Ipak, u praksi ćete, primjerice, željeti posjetitelja preusmjeriti na stranicu na kojoj će dobiti detaljnije informacije o toj narudžbi. U tom bi slučaju naredba u metodi izgledala, recimo, ovako:

```
Response.Redirect("WebForm2.aspx?id=" + e.Item.Cells[1].Text);
```

Kao što vidite, posjetitelj bi bio preusmjeren na drugu web-formu s parametrom koji sadrži tekst iz drugog (indeksi počinju od broja 0!) stupca. U našem je primjeru to stupac s identifikacijskim brojem narudžbe, što je točno onaj parametar koji se prosljeđuje u ovakvim situacijama.

Sortiranje zapisa

Vrlo je korisno omogućiti korisniku da prilikom pregledavanja sadržaja baze sortira određene stupce. Za posjetitelja je funkcionalnost jednostavna – klikne na ime stupca po kojem želi sortirati i očekuje rezultat. Što treba napraviti da bi mu se to omogućilo?



Slika 10-31: ***Tablica s mogućnošću sortiranja po drugoj i trećoj koloni***

U HTML-kôd kontrole treba dodati sljedeće:

```
<asp:DataGrid ... AllowSorting="True"> ...
```

Nadalje, kod definicije svake kolone po kojoj želimo uključiti sortiranje (unutar sekcije <Columns>) treba dodati ovaj parametar kojem ćemo kao vrijednost najčešće upisati ime polja u bazi:

```
<asp:BoundColumn DataField="OrderID" HeaderText="Broj narudžbe"
SortExpression="OrderID"></asp:BoundColumn>
```

To će nam osigurati promjene u sučelju koje su potrebne za sortiranje, a još nam preostaje napisati dva komadića kôda koji će samo sortiranje odraditi. Naime, prilikom sortiranja treba izmijeniti SQL--upit koji šaljemo bazi i zatim *rebindati* kontrolu. Stoga bi trebalo modificirati funkciju Bindanje() da, ovisno o parametru, doda parametar sortiranja u SQL-upit.

Mi nećemo brisati postojeću funkciju koja ne prima parametar (tako da ne moramo mijenjati dio kôda koji je koristi), nego ćemo napraviti još jednu preopterećenu inačicu s tim parametrom. Znači, uz postojeću funkciju Bindanje(), treba dodati sljedeće:

```
private void Bindanje(string sortiranje)
{
    SqlConnection sqlConn = new
        SqlConnection(ConfigurationSettings.AppSettings["connstr"]);
    SqlDataAdapter sqlComm = new SqlDataAdapter("SELECT * FROM Orders ORDER BY " +
        sortiranje, sqlConn);

    DataSet ds = new DataSet();
    sqlComm.Fill(ds);

    DataGrid1.DataSource = ds;
    DataGrid1.DataBind();
}
```

Evo i kako ćemo pozvati tu funkciju:

```
private void DataGrid1_SortCommand(object source,
    System.Web.UI.WebControls.DataGridSortCommandEventArgs e)
{
    Bindanje(e.SortExpression);
}
```

Naime, svojstvo e.SortExpression sadrži vrijednost koju smo definirali u HTML-kodu istoimenim parametrom. Drugim riječima, da smo tamo taj parametar definirali ovako:

```
SortExpression="OrderID DESC"
```

... narudžbe bi bile sortirane od one s najvećim brojem do one s najmanjim.

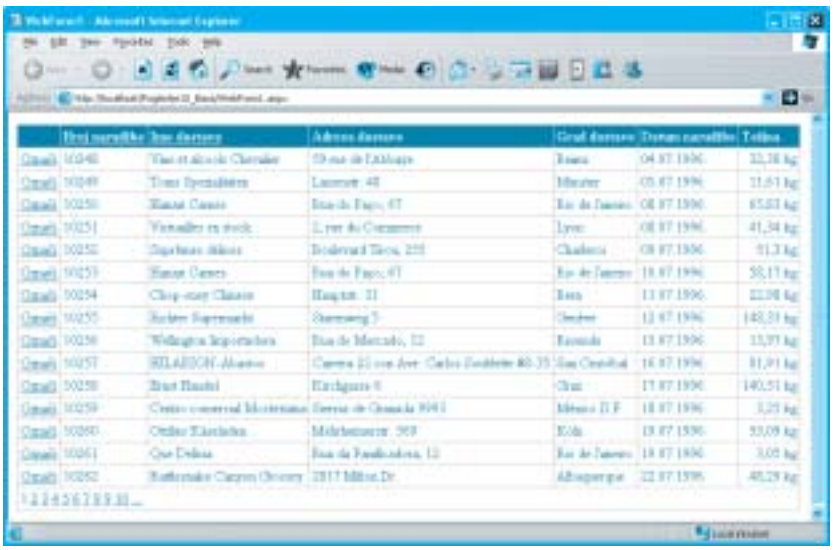
Pregled po stranicama

Ako se u bazi nalaze veće količine podataka, njihov prikaz na jednoj stranici rezultirao bi dugim učitavanjem i velikom nepreglednošću. Zato postoji mogućnost pregledavanja podataka po stranicama. U ASP.NET-u je to izuzetno jednostavno izvesti.

Kao i obično, za početak su potrebne određene modifikacije u HTML-kodu:

```
<asp:DataGrid ... AllowPaging="True" PageSize="15"> ...
```

Prvi parametar uključuje prikaz po stranicama, dok drugi određuje veličinu jedne stranice (broj zapisa po stranici).



Slika 10-32: ***Prikaz podataka iz baze po stranicama***

Naravno, i tu nam treba komadić kôda koji će spomenutu funkcionalnost odraditi. Ovoga se puta vežemo uz događaj PageIndexChanged:

```
private void DataGrid1_PageIndexChanged(object source,
    System.Web.UI.WebControls.DataGridPageChangedEventArgs e)
{
    DataGrid1.CurrentPageIndex = e.NewPageIndex;
    Bindanje();
}
```

Želite li umjesto brojeva stranica na dnu koristiti izraze prethodna i sljedeća stranica, dodajte unutar *tagova* kontrole sljedeći izraz:

```
<PagerStyle Mode="NextPrev" PrevPageText="prije"
NextPageText="poslije"></PagerStyle>
```

Imajte na umu da ovaj način prikazivanja podataka po stranicama nije i najbolji. Naime, stranica iz baze dobiva sve podatke, uključujući i one koji se ne prikazuju, što kod velikih količina podataka može biti relativno sporo.

Prilagodljivi stupci

Dosad smo u primjerima svako od polja iz baze trpali u zaseban stupac. To je često zgodno rješenje, no ponekad imamo potrebu, primjerice, spojiti više polja u isti stupac. U tome će nam pomoći prilagodljivi stupci (engl. *template column*).

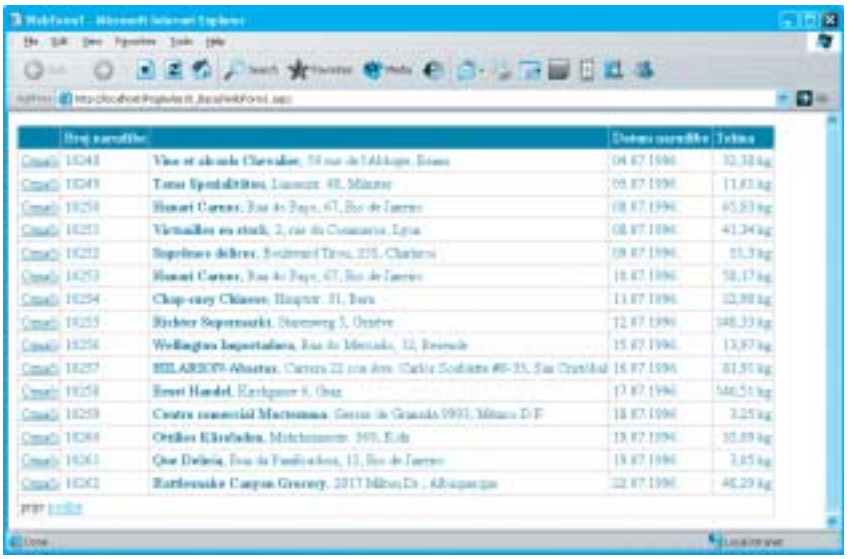
Te stupce definiramo na istom mjestu gdje smo definirali ostale vrste stupaca, u sekciji <Columns>:

```
<Columns>
   <asp:ButtonColumn ... ></asp:ButtonColumn>
   <asp:BoundColumn ... ></asp:BoundColumn>
   <asp:TemplateColumn ...>
      <!– definicija prilagodljivog stupca –>
   </asp:TemplateColumn>
</Columns>
```

Evo kako bi izgledao kôd za prilagodljivi stupac koji možete vidjeti na slici 10-33:

```
<asp:TemplateColumn>
   <ItemTemplate>
      <b><%# DataBinder.Eval(Container.DataItem, "ShipName") %></b>,
      <%# DataBinder.Eval(Container.DataItem, "ShipAddress") %>,
      <%# DataBinder.Eval(Container.DataItem, "ShipCity") %>
   </ItemTemplate>
</asp:TemplateColumn>
```

Predložak za stupac može sadržavati bilo koje HTML-elemente, pa čak i web-kontrole, i mora biti definiran između *tagova* <ItemTemplate> i </ItemTemplate>. Unutar predloška na pokazani način možemo pristupati i poljima u bazi te na taj način napraviti bilo kakav prikaz podataka.



Slika 10-33: ***Povezivanje više polja iz baze u isti stupac pomoću prilagodljivih stupaca***

Sada kad smo upoznali prilagodljive stupce, mogli bismo napisati nešto efikasniji način preusmjeravanja posjetitelja na novu stranicu. Sjetite se – nedavno smo tu stvar riješili pomoću stupca tipa ButtonColumn koji je radio *postback* te tamo radio Response.Redirect na željenu stranicu. Evo kako to napraviti pomoću prilagodljivog stupca jednostavnije i brže:

```
<asp:TemplateColumn>
   <ItemTemplate>
      <a href="WebForm2.aspx?id=<%# DataBinder.Eval(Container.DataItem, "OrderID")
%>">Detaljnije...</a>
   </ItemTemplate>
</asp:TemplateColumn>
```

Jednom kada napišete izraz poput ovoga, Visual Studio vam neće dozvoliti vratiti se u dizajnerski pogled stranice. Ne pitajte zašto, jednostavno neće. Problem rade dvostruki navodnici u izrazu, a tome možete doskočiti tako da stvar napišete na sljedeći način (uočite jednostruke navodnike na dva mjesta!):

```
<a href='WebForm2.aspx?id=<%# DataBinder.Eval(Container.DataItem,
"OrderID") %>'>Detaljnije...</a>
```

Ovaj problem se javlja samo kada navodnici obuhvaćaju referencu na polje u bazi.

Uređivanje zapisa

Vjerojatno se nećete iznenaditi kad vam kažemo da DataGrid omogućava i uređivanje zapisa. Postizanje ove funkcionalnosti slično je promjeni označenog retka na stranici, samo što dodajemo stupac tipa EditCommandColumn. Ovako:

```
<asp:EditCommandColumn EditText="Uredi" ButtonType="PushButton" UpdateText="Snimi"
CancelText="Odustani" />
```

Uočite da smo ovoga puta kao ButtonType stavili vrijednost PushButton (za razliku od LinkButton). To znači da će nam se u toj koloni pojaviti gumb, a ne link, iako bi oboje imali istu funkcionalnost.

I ovdje nam treba vezanje uz događaj, sad se događaj zove EditCommand, a pripadajuća funkcija mu izgleda ovako:

```
private void DataGrid1_EditCommand(object source,
    System.Web.UI.WebControls.DataGridCommandEventArgs e)
{
    DataGrid1.EditItemIndex = e.Item.ItemIndex;
    Bindanje();
}
```

Kada postavimo ovu vrijednost svojstvu EditItemIndex, redak u kojem smo kliknuli na gumb "Uredi" moći ćemo uređivati.

Poljima za koja ne želimo da budu uređivana (primjerice, identifikacijska) treba dodati parametar ReadOnly:

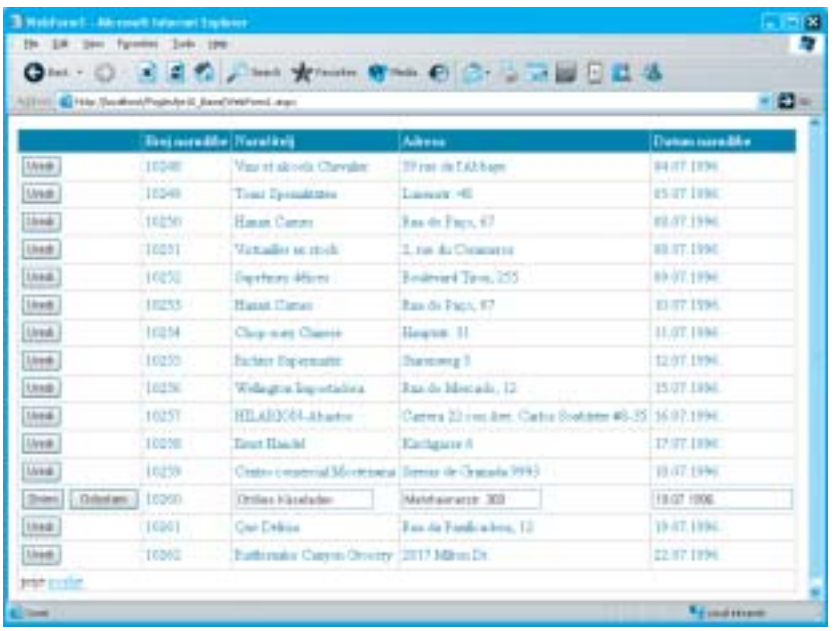
```
<asp:BoundColumn DataField="OrderID" SortExpression="OrderID" HeaderText="Broj
narudžbe" ReadOnly="True"></asp:BoundColumn>
```

Nakon što pokrenemo editiranje nekog retka, u njemu će se pojaviti dva nova gumba. Tekstove na tim gumbima definirali smo još na početku (parametri UpdateText i CancelText), a sada im treba napisati funkcionalnost.

Počnimo od jednostavnijega – gumb Cancel (odnosno, u našem primjeru, Odustani) ima zadatak isključiti uređivanje retka bez snimanja promjena. Stoga svojstvu EditItemIndex treba pridružiti

vrijednost -1 koja znači da se niti jedan redak ne uređuje. Kako prilikom klika na gumb Cancel nastaje događaj CancelCommand, sljedeću funkciju vežemo uz njega:

```
private void DataGrid1_CancelCommand(object source,
    System.Web.UI.WebControls.DataGridCommandEventArgs e)
{
    DataGrid1.EditItemIndex = -1;
    Bindanje();
}
```



Slika 10-34: ***Uređivanje zapisa u DataGridu***

S gumbom Update koji uključuje i snimanje promjena imamo nešto više posla. Sljedeću funkciju vezat ćemo uz događaj UpdateCommand (funkciju ćemo sjeći komentarima, no to je sve jedna funkcija):

```
private void DataGrid1_UpdateCommand(object source,
    System.Web.UI.WebControls.DataGridCommandEventArgs e)
{
    // spremanje vrijednosti iz polja obrasca u varijable
    int OrderID = Convert.ToInt32(e.Item.Cells[1].Text);
```

U varijablu OrderID spremamo vrijednost iz drugog stupca (index 1). Kako smo to polje označili kao ReadOnly, u njemu se nalazi samo identifikacijski broj zapisa kojem pristupamo svojstvom Text. Naravno, kako se radi o broju, konvertiramo ga u *int*.

```
TextBox ShipNameText = (TextBox)e.Item.Cells[2].Controls[0];
TextBox ShipAddressText = (TextBox)e.Item.Cells[3].Controls[0];
TextBox OrderDateText = (TextBox)e.Item.Cells[4].Controls[0];

string ShipName = ShipNameText.Text;
string ShipAddress = ShipAddressText.Text;
DateTime OrderDate = DateTime.ParseExact(OrderDateText.Text, "dd.MM.yyyy.",
    null);
```

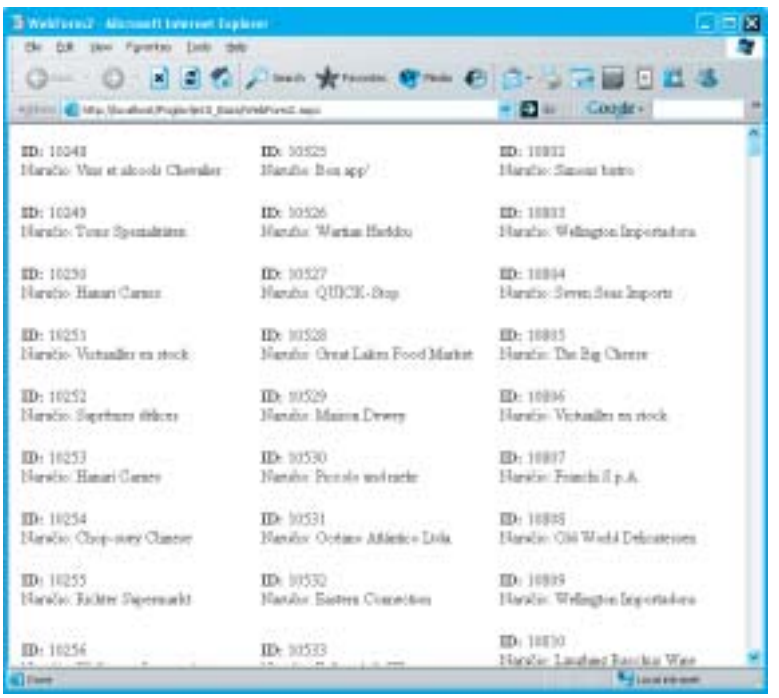
Kod ostalih polja stvar je nešto složenija. Naime, unutar stupca ne nalazi se čisti tekst, već kontrola tipa TextBox. Kako je ona jedina kontrola u stupcu, pristupamo joj preko kolekcije kontrola uz indeks 0 (Controls[0]). Mi smo sigurni da se radi o kontroli tipa TextBox, no kompajler nije, pa radimo *castanje*. Zatim je lako doći do vrijednosti unutar kontrole (svojstvo Text). Kako je svojstvo Text tipa *string*, kod datuma smo morali raditi konverziju pomoću metode ParseExact.

```
// definiranje SQL-upita za Update
string sqlUpit = "UPDATE Orders SET ShipName = @ShipName, ShipAddress =
   @ShipAddress, OrderDate = @OrderDate WHERE OrderID = @OrderID";

// stvaranje konekcije prema bazi
SqlConnection sqlConn = new
   SqlConnection(ConfigurationSettings.AppSettings["connstr"]);

// otvaranje konekcije prema bazi
sqlConn.Open();

// kreiranje SQL-naredbe
SqlCommand sqlComm = new SqlCommand(sqlUpit, sqlConn);

// dodavanje parametara
sqlComm.Parameters.Add("@OrderID", OrderID);
sqlComm.Parameters.Add("@ShipName", ShipName);
sqlComm.Parameters.Add("@ShipAddress", ShipAddress);
sqlComm.Parameters.Add("@OrderDate", OrderDate);

// izvršavanje SQL-upita
```

```
        sqlComm.ExecuteNonQuery();

        // zatvaranje konekcije prema bazi
        sqlConn.Close();

        // isključivanje uređivanja retka
        DataGrid1.EditItemIndex = -1;
        Bindanje();
    }
```

Ostatak kôda već smo susreli u prošlom poglavlju, pa nećemo ulaziti u detalje.

Gornji primjer nije otporan na gluposti koje posjetitelj može upisati u polja. Primjerice, može upisati tekst duži od predviđenog u bazi podataka ili pak unijeti krivi format datuma. Drugim riječima, dobro bi došle provjere valjanosti upisanih podataka.



Slika 10-35: ***Prikaz podataka u tri stupca pomoću kontrole DataList***

Oprez! Uljezi u upitu!

Gornji smo primjer mogli riješiti i malo drugačije. Primjerice, umjesto korištenja parametara u SQL-upitu, mogli smo napisati nešto poput ovoga:

```
string sqlUpit = "UPDATE Orders
SET ShipName = '" + ShipName +
"' WHERE OrderID = " + OrderID;
```

Međutim, takvo prosljeđivanje parametara SQL--upitu može biti kobno! O čemu se radi? Uzmimo za primjer da je posjetitelj naših stranica upisao u tekstualno polje prilikom editiranja sljedeću vrijednost:

```
'; DELETE FROM Orders; UPDATE
Orders SET ShipName = '
```

Vrlo naivno, računalo taj unos smatra legitimnim i ubacuje ga u naš SQL-upit, koji ispada ovakav:

```
UPDATE Orders SET ShipName = '';
DELETE FROM Orders; UPDATE
Orders SET ShipName = '' WHERE
OrderID = 12345
```

Zlobni posjetitelj je trikom naš upit pretvorio u tri upita od kojih je jedan od njih koban jer će izbrisati sve zapise tablice na koju se odnosi!

Ovakav napad naziva se *SQL injection*, a može ga se izbjeći jednostavnom konverzijom svakog jednostrukog navodnika koji je posjetitelj unio u dva jednostruka navodnika. Međutim, puno je bolji način koji smo pokazali u primjeru uz tekst – parametri se po službenoj dužnosti brinu o takvim, ali i brojnim drugim sitnicama.

Kontrola DataList

Najkraće rečeno, DataList je kontrola većih mogućnosti prilagodbe od DataGrida, no zato neke funkcionalnosti traže više podešavanja i programiranja.

Evo primjera kojem ćemo prikazati neka polja tablice Orders kao na slici 10-35:

```
<asp:DataList id="DataList1" runat="server" RepeatColumns="3">
   <ItemTemplate>
      <b>ID:</b> <%# DataBinder.Eval(Container.DataItem, "OrderID") %>
      <br />
      Naručio: <%# DataBinder.Eval(Container.DataItem, "ShipName") %>
      <p />
```

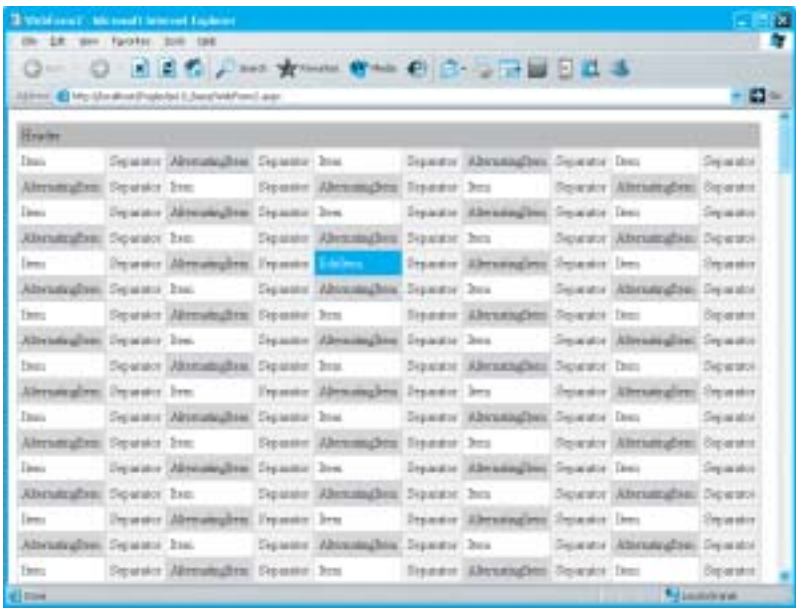
```
    <ItemTemplate>
</asp:DataList>
```

Da bi se kontrola prilikom izvršavanja stranice popunila podacima, potrebno je napraviti povezivanje s bazom na isti način koji smo to radili za kontrolu DataGrid metodom Bindanje().

Osim osnovnog predloška za zapis, kontrola DataList može sadržavati i neke druge predloške.

Predložak AlternatingItemTemplate, ukoliko je definiran, koristit će se za prikaz svakog drugog zapisa. U kombinaciji s ItemTemplateom možete postići da se zapisi naizmjence prikazuju koristeći jedan pa drugi predložak.

Pomoću predložaka HeaderTemplate i FooterTemplate definiramo zaglavlje i podnožje tablice s podacima. Imajte na umu da se oni neće vidjeti na stranici, ako su isključena svojstva ShowHeader odnosno ShowFooter.



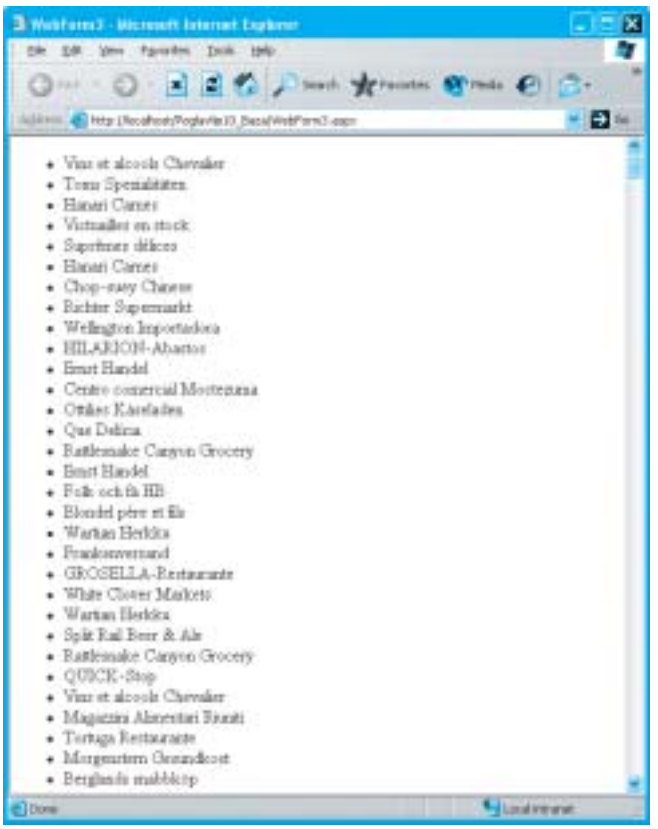
Slika 10-36: ***Demonstracija predložaka u kontroli DataList***

EditItemTemplate služi za definiranje predloška za zapis koji se uređuje (što nam pruža puno više mogućnosti od DataGrida), dok je SeparatorTemplate predložak za polje koje će se pojavljivati između zapisa.

Kontrola Repeater

Za one željne maksimalne kontrole nad prikazom podataka postoji kontrola Repeater. Ona nema nikakvih kozmetičkih svojstava, stilova za predloške niti zapise smješta u tablice ili *tagove* (ovo potonje može napraviti samo kontrola DataList ako joj promijenite svojstvo RepeatLayout). Kod nje je prikaz podataka u potpunosti u rukama programera.

Kontrola Repeater i DataList rade pomoću istih predložaka, no s nešto ograničenijim brojem mogućnosti.



Slika 10-37: ***Prikaz podataka iz baze pomoću kontrole Repeater***

Evo primjera u kojem ćemo zapise iz baze prikazati kao na slici 10-37 (niti tu nemojte zaboraviti metodu Bindanje):

```
<asp:Repeater id="Repeater1" runat="server">
   <HeaderTemplate>
```

```
        <ul>
    </HeaderTemplate>
    <ItemTemplate>
        <li><%# DataBinder.Eval(Container.DataItem, "ShipName")%></li>
    </ItemTemplate>
    <FooterTemplate>
        </ul>
    </FooterTemplate>
</asp:Repeater>
```

Na milost posjetiteljima

Na kraju razvoja svake web-aplikacije doći ćete na zadnju stepenicu – prebacivanje aplikacije na poslužitelj. To će uglavnom biti izuzetno jednostavan posao.

Prije prebacivanja treba pripremiti poslužitelj – otvoriti novi web-*site* ili kreirati novu web-aplikaciju, podesiti DNS i eventualno IP-adrese.

Zatim ide prebacivanje web-aplikacije na poslužitelj, što se svodi na kopiranje datoteka. Trebat ćete prebaciti sve konfiguracijske datoteke, datoteke s HTML-kôdom (".aspx", ".ascx", ".asax"...) te mapu "bin". U njoj se nalazi datoteka s nastavkom ".dll" u kojoj je kompajlirani kôd iz svih datoteka s pozadinskim kôdom (".aspx.cs", ".ascx.cs", ".asax.cs"...) tako da njih nije potrebno prebacivati na poslužitelj.

Prije prebacivanja DLL datoteke na poslužitelj treba je kompajlirati u načinu *realese* (vidi kraj osmog poglavlja).

Na produkcijskom serveru ćete vjerojatno koristiti i neki drugi izvor baze podataka pa će vjerojatno biti potrebno prilagoditi (zato je vrlo dobra praksa smještati ga u konfiguracijsku datoteku).

Dodatne komponente koje ste eventualno koristili također se nalaze u DLL-datotekama u mapi "bin". I njih je potrebno kopirati na poslužitelj.

No, naravno, osnovni uvjet i dalje postoji – poslužitelj mora imati instaliran .NET Framework.

11. POGLAVLJE

XML

U ovom poglavlju:

- Osnovni pojmovi vezani uz XML
- Rad s XML-om u .NET-u
- Dodavanje, traženje, mijenjanje i brisanje dijelova XML dokumenata
- Učitavanje, spremanje i pisanje XML dokumenata
- Osnove XPatha
- Pretraživanje XML dokumenata XPathom
- XML serijalizacija

XML (eXtensible Markup Language) posljednjih nekoliko godina možete pronaći u reklamnim materijalima gotovo svake aplikacije – svi se hvale da mogu izvoziti svoje datoteke i spremati ih u XML formatu da interno koriste XML, kao da je to nešto što čini njihovu aplikaciju moćnijom od konkurencije. U ovom poglavlju ćemo razjasniti što je uopće XML, kako se on može uklopiti u vašu aplikaciju te gdje zapravo leži snaga XML-a.

Naravno, i Microsoft je uvidio važnost XML formata i njegova podrška je uključena u niz programa (primjerice, Microsoft Office ima mogućnost spremanja svih datoteka u XML formatu). Ipak, kako se radi o formatu koji će pretežito koristiti programeri u svojim aplikacijama i za nadogradnju postojećih rješenja, podrška za XML u Visual Studiju .NET je na svakom koraku.

Osnovni pojmovi

Dakle, mnogo je bilo priče o XML-u kao spasitelju informatičke industrije, nečemu što će poboljšati sve aplikacije, učiniti život programera lakšim i učiniti sve kupce softvera zadovoljnima. Očito se radi o pretjeranim tvrdnjama jer je takvih tehnologija "spasitelja" kroz povijest bilo mnogo, no ipak – XML je po mnogočemu poseban i može vašem programu i načinu razmišljanja predstavljati spas.

Što je uopće XML? Radi se o običnom tekstualno strukturiranom formatu zapisa. No dobro, možete pomisliti, što je tu toliko moćno? Upravo ta jednostavnost postavlja XML na uzvišenu poziciju.

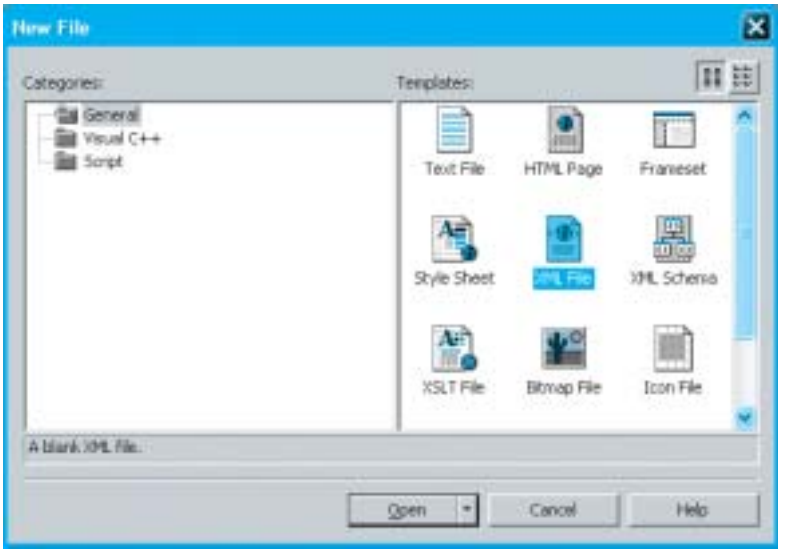
XML format je zapravo preporuka World Wide Web Consortiuma (W3C, više informacija na http://www.w3c.org/), grupe koja je zadužena za donošenje različitih standarda i koja je, primjerice, već prije definirala HTML, CSS, a zadužena je i za druge tehnologije vezane uz XML koje će biti objašnjene u ovom poglavlju, poput XPatha i XSLT-a..

XML format ćete koristiti za spremanje različitih podataka, konfiguracijskih datoteka, kao jedan od izlaznih formata za vaše datoteke, kao format za podatke koje ćete razmjenjivati s drugim aplikacijama. Kako se radi o tekstualnom tipu podatka, on je čitljiv na svim platformama. Čitati i proučavati ga mogu i korisnici, tako da ga otvore u bilo kojem uređivaču teksta.

XML se sastoji od hijerarhijskih elemenata, baš kao i HTML (npr. HTML *tag* sadržava HEAD *tag*, a on pak može sadržavati TITLE i META *tagove*). No dok HTML ima točno određene elemente koje može sadržavati, XML je mnogo slobodniji – u njemu možete definirati koju god strukturu želite. Za XML se kaže da je proširiv, što u praksi znači da može imati koliko želite različitih elemenata, na kojim god pozicijama u dokumentu odnosno da vi sami određujete format svog XML dokumenta.

Krenimo odmah i na konkretan primjer – već smo objasnili da je uloga XML-a da sadržava neke podatke pa ćemo sad izraditi jednostavnu XML datoteku koja će sadržavati informacije o kupcima u nekom dućanu.

Da biste stvorili novu XML datoteku, u Visual Studiju .NET odaberite *File – New – File* ili pritisnite CTRL+N. Pojavit će vam se prozor kao na slici 11-1, a u njemu odaberite stvaranje XML datoteke.



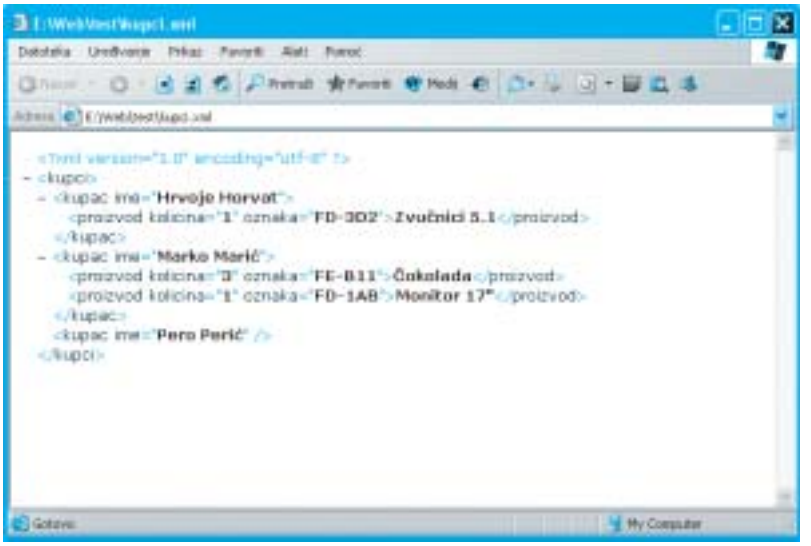
Slika 11-1: ***Stvaranje nove XML datoteke u Visual Studiju .NET***

U njoj će već biti upisano zaglavlje dokumenta, a vi možete upisati ostatak sadržaja. Evo primjera XML dokumenta.

```
<?xml version="1.0" encoding="utf-8" ?>
<kupci>
   <kupac ime="Hrvoje Horvat">
      <proizvod kolicina="1" oznaka="FD-3D2">Zvučnici 5.1</proizvod>
   </kupac>
   <kupac ime="Marko Marić">
      <proizvod kolicina="3" oznaka="FE-B11">Čokolada</proizvod>
      <proizvod kolicina="1" oznaka="FD-1AB">Monitor 17"</proizvod>
   </kupac>
   <kupac ime="Pero Perić" />
</kupci>
```

Pogledate li primjer, uočit ćete da je njegova struktura veoma slična HTML-u. Glavnu ulogu imaju elementi, odnosno *tagovi*, koji pak imaju neke atribute, a svi oni su složeni hijerarhijski.

Također, specifičnost XML formata za podatke je u tome što je on svima lako razumljiv. Gornji primjer zaista ne skriva neke tajne – sadržava informacije o kupcima koji su složeni hijerarhijski ispod glavnog *taga* nazvanog "kupci". Za svakog kupca zapisano je i njegovo ime te proizvodi koje je kupio (logika je i dalje ista, pa je za svaki proizvod zapisana njegova količina i oznaka na skladištu te opis). Uočite da smo sami imenovali sve *tagove* i odredili koje podatke mogu sadržavati.



Slika 11-2: ***Internet Explorer će na pregledan način prikazati strukturu XML dokumenta.***

Prethodni primjer ima i jednu posebnost. Naime, pogledate li kupca Peru Perića, vidjet ćete da za njega nema nikakvih informacija o proizvodima, vjerojatno zato što nijedan nije kupio. No posebnost se očituje u načinu zatvaranja njegovog taga "kupac". Dok se može očekivati da kao i u HTML-u postoji obavezno "</kupac>", u slučaju da tag ništa ne sadržava, može ga se zatvoriti i na kraći način – zatvaranjem taga sa znakovima "/>".

XML dokumenti su obične tekstualne datoteke, pa ih zato možete izrađivati u bilo kojem tekstualnom uređivaču, od Notepada do Visual Studija. Naravno, preporučujemo vam da ostanete uz Visual Studio jer on ima ugrađenu podršku za XML format pa vam može pomoći u pisanju – automatski će zatvarati sve *tagove* koje napišete.

Preporučujemo vam da konfiguracijske datoteke, kao uostalom i sve datoteke koje koristite u programiranju, otvarate i mijenjate pomoću Visual Studija. Naime, sve se datoteke prema inicijalnim postavkama spremaju u formatu Unicode, što mnogi editori ne podržavaju. Nakon snimanja datoteka s takvim programima, one bi mogle ispasti neispravne i neupotrebljive.

Kad budete radili s malo složenijim XML dokumentima možda ćete se i teže u njima snalaziti. XML format ne zahtijeva od vas da uvlačite sve elemente koji su niže po hijerarhiji i tako si olakšavate pregled. Što se samog formata tiče, *tagove* možete naslagati jedan na drugi, no pokušajte se tada snaći u sadržaju.

Na svu sreću, Internet Explorer u potpunosti podržava prikaz XML datoteka. Otvorite načinjenu datoteku u Internet Exploreru i vidjet ćete hijerarhijsku strukturu dokumenta. Čak možete i zatvarati podstrukture nekih elemenata klikom na znak "-" kraj njega.

Vi imate potpunu slobodu u radu s XML-om. Na vama je da osmislite format dokumenata, napunite ga podacima i izradite programsku podršku koja će iskoristiti XML datoteke (to vas čeka u nastavku poglavlja). No ponekad baš ta sloboda može biti ograničavajući faktor. Ponekad možete podatke prezentirati i oblikovati na krivi način te oni mogu biti nerazumljivi, teški za obradu, višeznačni i slično.

Za XML je presudno da ga svi mogu razumjeti. Ljudi bi ga trebali moći čitati i shvatiti okvirno o čemu se u tom dokumentu radi, a računala i programi bi trebali moći obraditi ga i iz njega izvući jednoznačne podatke. Primjerice, pogledajte neispravno korištenje XML-a:

```
<?xml version="1.0" encoding="utf-8" ?>
<proizvodi>
   <proizvod oznaka="FD-3D2">
      <kolicina vrijednost="1">
         <kupac ime="Hrvoje Horvat">Zvučnici 5.1</kupac>
      </kolicina>
   </proizvod>
   <proizvod oznaka="FE-B11">
      <kolicina vrijednost="3">
         <kupac ime="Marko Marić">Čokolada</kupac>
      </kolicina>
   </proizvod>
   <proizvod oznaka="FD-1AB">
      <kolicina vrijednost="1">
         <kupac ime="Marko Marić">Monitor 17"</kupac>
      </kolicina>
   </proizvod>
</proizvodi>
```

Iako će to biti objašnjeno kasnije u tekstu, nije naodmet ponoviti – struktura gornjeg primjera nije korisna jer nelogično prikazuje podatke. U kasnijem tekstu ćemo se više puta referencirati na primjer XML dokumenta, a pritom ćemo misliti na njegovu prvu, logičnu verziju, a ne na ovu.

Na prvi je pogled i ovaj dokument potpuno jasan. Organiziran je po proizvodima, a za svaki je proizvod zapisana količina i koji je kupac tu količinu kupio. Na kraju cijele hijerarhije nekog proizvoda zapisan je i sam opis tog proizvoda. Naravno, možda se to moglo i bolje organizirati, no u žurbi ste, a ovaj oblik je zadovoljavajući. Ni ne slutite potencijalne probleme.

No oni su vrlo stvarni – kako ćete znati da kupac Pero Perić nije ništa kupio? Zamislite da se radi o malo većem dokumentu i pokušajte na prvi pogled saznati što je sve kupio Marko Marić. Okvirno, ovaj oblik XML dokumenta nije toliko loš (organiziran je po proizvodima), no potpuno je kriva hijerarhija u kojoj *kupac* dolazi ispod *količine*. Naravno, besmisleno je i da se opis proizvoda nalazi unutar elementa *kupac*.

XML pravila

Iako vam se može činiti da je XML doista slobodan format, postoji nekoliko pravila kojih se morate pridržavati da bi XML dokument bio valjan:

1. Mora postojati jedan i samo jedan korijenski (glavni) element.

2. Svi *tagovi* se moraju zatvoriti.

3. Imena elemenata su osjetljiva na velika i mala slova.

U našem prvom primjeru postoji i <?xml version="1.0" encoding="utf-8" ?> element, no on nije obavezan i njegova je svrha opisna. Ukoliko ga izbacite, dokument će i dalje biti ispravan.

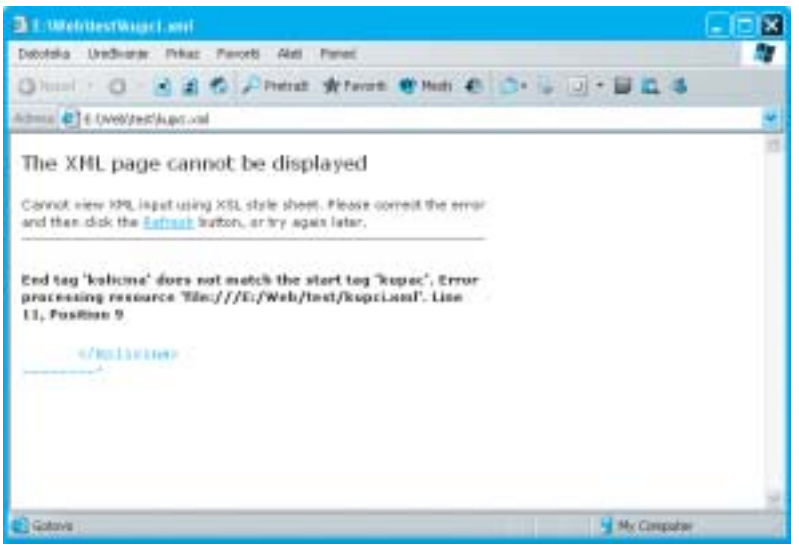
Prvo pravilo kaže da mora postojati jedan i samo jedan glavni element, a to je u našem slučaju element *kupci*. Unutar njega su sadržani svi ostali elementi. Ukoliko njega ne bi bilo, svi bi njegovi podelementi *kupac* bili glavni, a to je nedozvoljeno, jer kao što pravilo kaže – mora postojati *jedan i samo jedan*.

Drugo pravilo govori o zatvaranju *tagova*. To je već objašnjeno – *tagove* možete zatvoriti standardnim, HTML, načinom tako da stavite krajnji *tag* oblika "</ ... >", no možete ih i odmah zatvoriti tako da na kraj *taga* stavite znakove "/>".

Treće pravilo može biti problematično ukoliko ručno u Notepadu pišete XML dokument. Primjerice, greška je ako napišete "<kupacIme>Hrvoje Horvat</kupacime>", jer *kupacIme* nije isto što i *kupacime*.

XML parseri, odnosno programi zaduženi za obradu XML dokumenata, ne znaju ništa raditi s dokumentima koji nisu ispravni. Ukoliko se vaš dokument ne podvrgava spomenutim pravilima, ne smatra ga se XML dokumentom. Pokušajte neku od opisanih grešaka napisati u XML-u, te zatim taj dokument učitati u Internet Exploreru. Kao što je i očekivano, dojavit će vam grešku.

No, da ne duljimo previše, veoma je važna ispravna organizacija XML dokumenata. Nakon što izradite svoj XML, postavite si osnovna pitanja o njegovu korištenju. Ključno je da razumijete osnovni princip – ako vi možete na jednostavan način izvući neke podatke, tad će to moći i program. Ukoliko vi morate pamtiti više stvari, puno *scrollati* po dokumentu da biste isti podatak izvukli s više mjesta, to očito nije dobro jer će istu stvar morati raditi i program, a to će biti sporo i loše.



Slika 11-3: ***Internet Explorer neće otvoriti neispravan XML dokument, već će vam dojaviti grešku.***

Da biste bili sigurni u ispravnost XML dokumenata, iskoristite mogućnost njihova provjeravanja direktno iz Visual Studija .NET. Odaberite opciju XML – Validate XML Data, a u prozoru Task List će se pojaviti popis svih grešaka i upozorenja s detaljnim informacijama.

XML DOM

Uz XML, W3C se pobrinuo i za definiranje standardiziranog API-ja nazvanog XML DOM (od Document Object Model). DOM API predstavlja XML dokument kao stablo elemenata – zahvaljujući njegovoj hijerarhijskoj strukturi, lako je posložiti sve elemente i predstaviti ih u obliku *stabla* u kojem postoje korijenski element te elementi *listovi* (koji nemaju više svoje *djece* odnosno podelemenata).

Kretanjem po stablu lako je pronaći bilo koji element u XML strukturi. Predočite li hijerarhijsku strukturu elemenata tako da svaki element može imati svoju *djecu* odnosno elemente koji su u

hijerarhiji ispod njega te *roditelja* odnosno element koji ga sadržava, lako se snalaziti razmišljajući o odnosima poput *djed* (dva elementa iznad) ili *unuk* (dva elementa ispod).

API je skraćenica od Application Program Interface i predstavlja definirane metode sučelja koje se mogu koristiti za rad s programom.

Vratimo li se na originalan ispravan primjer XML dokumenta, možete uočiti da su *unuci* elementa "kupci" zapravo elementi "proizvod" jer se nalaze dvije razine ispod njega. Isto tako, *roditelj* elementa "proizvod" je element "kupac". Razmišljate li na taj način, vrlo lako ćete se snalaziti u XML strukturama.

Ukoliko vas zanima kompletna i detaljna specifikacija XML DOM-a, možete je pronaći na http://www.w3.org/DOM.

DOM API vam pruža metode za rad s XML dokumentima pa tako standardiziranim metodama možete pronaći korijenski element u XML dokumentu, popis svih elemenata određenog imena, popis svih podelemenata, tj. *djece* nekog elementa, vrijednost pojedinog atributa nekog elementa itd.

Osim dohvaćanja informacija iz XML dokumenta, korištenjem DOM API-ja možete i dodavati, mijenjati ili brisati elemente iz dokumenta, a možete i dodavati, mijenjati i brisati sve njihove atribute.

.NET u sebi ima ugrađen XML DOM API te ćemo njega koristiti u kasnijim primjerima baš za pokazivanje prethodno navedenih mogućnosti.

Uz DOM način rada s XML dokumentima postoji još jedan standard – SAX ili Simple API for XML. Razlika između DOM-a i SAX-a je u načinu njihova rada – dok DOM parseri koji obrađuju XML dokument učitavajući ga cijelog u memoriju računala (što može usporiti aplikaciju ili čak biti nemoguće za ekstremno velike XML datoteke), SAX pruža način slijednog čitanja sadržaja XML dokumenta bez potrebe da ga cijelog učitava u memoriju. Pritom se služi *callback* metodama koje se pozivaju pri određenim događajima, primjerice pri pojavi određenog tipa elementa. Iako SAX nije podržan u .NET-u, ipak postoji klasa *XmlReader* koja na sličan način obrađuje XML dokumente – slijednim čitanjem unaprijed.

XPath

I dok DOM ima izvrsne mogućnosti dohvaćanja određenih dijelova XML dokumenta, ipak ima određena ograničenja za pretraživanje. No to nije problem – uz XML tehnologiju se veže i XPath (od XML Path), tehnologija za slobodno pretraživanje XML dokumenata.

XPath je vrlo sličan SQL upitima. Naravno, dok SQL upiti pretražuju baze podataka, XPath pretražuje XML dokumente i služi isključivo za dohvaćanje njegovih dijelova, dakle poput naredbe SELECT.

XPath upiti tako mogu sadržavati različite uvjete, koristiti logičke operatore, funkcije za rad sa znakovnim nizovima (i tako, primjerice, pronaći sve elemente koji u svom nazivu sadrže niz "abc"), aritmetičke operatore (primjerice, možete pronaći sve elemente čiji je zbroj neka dva atributa veći od 100) i slično.

Naredni XPath upit iz našeg će primjera XML dokumenta dohvatiti sve elemente imena "proizvod" čiji je atribut oznaka jednak "FE-B11", ma gdje se oni nalazili u XML strukturi.

```
//proizvod[@oznaka='FE-B11']
```

Zanimaju li vas detalji XPath specifikacije odnosno nešto više od onoga što ćemo obraditi u ovom poglavlju, proučite dokumentaciju na web-stranici http://www.w3.org/TR/xpath.

XSLT

Bavite li se izradom web-stranica, poznat vam je koncept po kojem se sadržaj web-stranica odvaja od informacija o njenom prikazu. Konkretno, za sadržaj web-stranica služe HTML datoteke, a u CSS datotekama se nalaze informacije o prikazu pojedinih dijelova sadržaja.

Na sličan način se može gledati i na odnos XML-a i XSLT-a – dok XML sadržava podatke, XSLT sadržava upute za njihovo oblikovanje i prikaz. XSLT (eXtensible Stylesheet Language Transformations) je dakle format za zapisivanje transformacija XML dokumenata. Korištenjem različitih XSLT datoteka moguće je istu XML datoteku prikazati na potpuno drugačiji način.

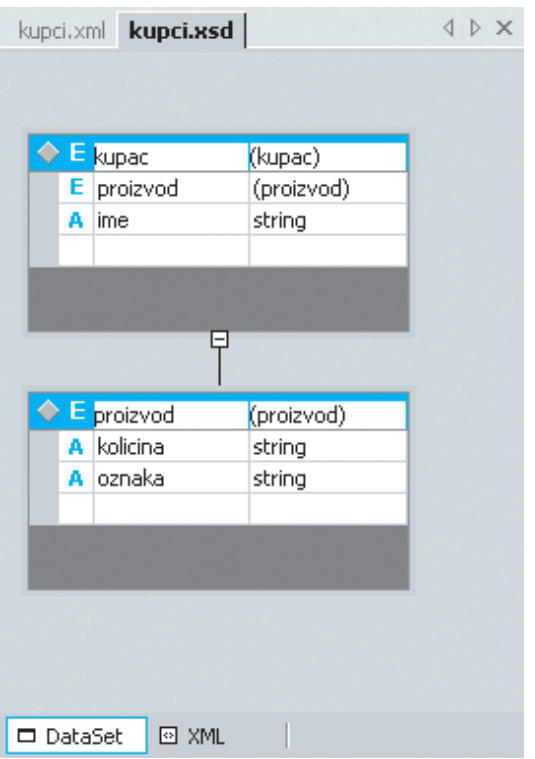
Proučavajući XML dokumentacije često ćete naići na kraticu XSL (eXtensible Stylesheet Language). Radi se o kombinaciji triju W3C specifikacija – XPatha, XSLT-a i XSL-FO (XSL Formatting Objects).

XML sheme

XML je izvrstan format za razmjenu podataka – vi podatke iz svoje aplikacije pretvorite u neki XML oblik i proslijedite ga autoru neke druge aplikacije na korištenje i izradu podrške za taj format. No podaci iz XML dokumenta nisu dovoljni za razumijevanje strukture XML dokumenta – tome služe XML sheme. One opisuju strukturu dokumenta i na temelju njih moguće je točno znati kakvu će XML strukturu imati neki dokument.

Preteča XML shema imala je ime DTD (Document Type Definition) i služila je za provjeru ispravnosti XML dokumenta, no ipak s manjim mogućnostima nego što imaju XML sheme. DTD-ovi se i danas još uvijek koriste, pa je podrška i za njih ugrađena u .NET.

Shemu odnosno opis nekog XML dokumenta možete izraditi automatski u Visual Studiju. Učitajte XML dokument i odaberite opciju *XML – Create Schema*.



Slika 11-4:
Stvorena shema za XML dokument

Stvorite li tako shemu za naš XML primjer, dobit ćete XSD datoteku sljedećeg sadržaja, a njen grafički prikaz možete vidjeti na slici 11-4.

```
<?xml version="1.0"?>
<xs:schema id="kupci" targetNamespace="http://tempuri.org/kupci1.xsd"
xmlns:mstns="http://tempuri.org/kupci1.xsd" xmlns="http://tempuri.org/kupci1.xsd"
xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:msdata="urn:schemas-microsoft-
com:xml-msdata" attributeFormDefault="qualified" elementFormDefault="qualified">
  <xs:element name="kupci" msdata:IsDataSet="true" msdata:Locale="hr-HR"
      msdata:EnforceConstraints="False">
    <xs:complexType>
      <xs:choice maxOccurs="unbounded">
        <xs:element name="kupac">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="proizvod" nillable="true" minOccurs="0"
                  maxOccurs="unbounded">
                <xs:complexType>
                  <xs:simpleContent msdata:ColumnName="proizvod_Text"
                      msdata:Ordinal="2">
                    <xs:extension base="xs:string">
                      <xs:attribute name="kolicina" form="unqualified"
                          type="xs:string" />
                      <xs:attribute name="oznaka" form="unqualified"
                          type="xs:string" />
                    </xs:extension>
                  </xs:simpleContent>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
            <xs:attribute name="ime" form="unqualified" type="xs:string" />
          </xs:complexType>
        </xs:element>
      </xs:choice>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Obratite pozornost na bitne elemente – s *xs:element* su označeni zasebni elementi u XML strukturi, a za svaki od njih su namještene i druge njegove postavke. Kad se dogovorite oko sheme XML

dokumenta, više ne može biti zabune – lako možete provjeriti odgovara li neki XML dokument zadanoj shemi i tako možete biti sigurni da će svi programi za rad s određenim tipom XML dokumenta raditi sa svim XML datotekama koje slijede zadanu shemu.

Detaljne specifikacije XML shema možete pronaći na web-adresi http://www.w3.org/XML/Schema.

Podrška XML-u u .NET-u

Kao što ste vidjeli, sve dosad opisane tehnologije i formate standardizirao je W3C. Dakle, ništa od toga nije isključivo Microsoftovo – svoje znanje o XML-u, XPathu, XSLT-u ili XML shemama možete primijeniti na bilo kojoj platformi i u bilo kojem programskom jeziku.

U .NET-u postoji ugrađena podrška za rad sa svim opisanim tehnologijama. Kao i sve ostalo, ona je sadržana u nizu klasa, čiji popis možete vidjeti u tablici 11-1.

Kao što vidite, većina klasa organizirana je ispod *System.Xml namespacea*, što olakšava snalaženje.

Tablica 11-1:
Popis glavnih klasa u .NET-u za rad s XML tehnologijama

Klasa	Opis
System.Xml.XmlReader	apstraktna klasa za čitanje XML dokumenata
System.Xml.XmlTextReader	klasa koja pruža najbrži način čitanja XML dokumenta; po funkcionalnosti je slična *DataReaderu* u ADO.NET-u jer pruža mogućnost čitanja samo unaprijed (ne možete se vraćati) i pomoću nje ne možete mijenjati sadržaj XML dokumenta; ova klasa nema mogućnost validiranja XML dokumenta
System.Xml.XmlValidatingReader	klasa za čitanje XML dokumenata koja ima mogućnost validiranja njegove ispravnosti prema XML shemama i DTD dokumentima
System.Xml.XmlNodeReader	čitač XML dokumenta za DOM elemente
System.Xml.XmlWriter	apstraktna klasa za pisanje XML dokumenata
System.Xml.XmlTextWriter	implementacija klase za pisanje XML dokumenata koja ima mogućnost stvaranja XML dokumenta samo unaprijed i predstavlja najbrži način pisanja XML dokumenata

Klasa	Opis
System.Xml.XmlDocument	klasa koja predstavlja XML dokument
System.Xml.XmlDataDocument	implementacija klase za rad s XML dokumentom koja pruža mogućnost pristupa podacima relacijskim načinom ili preko DOM-a
System.Xml.XPath.XPathDocument	klasa za XML dokument optimizirana za zadavanje XPath upita
System.Xml.XPath.XPathNavigator	omogućava XPath upravljanje XML dokumentom
System.Xml.Schema.XmlSchema	klasa za XSD shemu XML dokumenta
System.Xml.Xsl.XslTransform	klasa za XSL transformacije

Rad s XML-om

Naučiti raditi s XML-om u .NET-u najlakše je na temelju primjera. Stoga ćemo pokazati kako napisati kôd koji bi stvorio XML dokument te dodavao, mijenjao i brisao elemente. Kako XML možete koristiti u svim tipovima aplikacija, naredni kôd neće biti vezan niti uz jednu vrstu aplikacija, već ćete ga moći koristiti i u prozorskim aplikacijama, web-aplikacijama ili web-servisima.

Osim glavnih klasa prikazanih u tablici 11-1, u *namespaceu System.Xml* postoji još niz klasa za rad s XML dokumentima. Njih ćemo koristiti u narednim primjerima, a pobrojane su u tablici 11-2.

Tablica 11-2:
Klase za rad s dijelovima XML dokumenta

Klasa	Opis
XmlNode	jedan čvor (element, komentar, CDATA dio, itd.) u XML dokumentu
XmlNodeList	lista objekata XmlNode; kretanje po XML dokumentu zasniva se na prolaženju kroz stablo XmlNodeList, jer svaki XML element može sadržavati sve svoje podelemente (koji se opet daju predstaviti novim objektom XmlNodeList)
XmlDocument	XML dokument; ova klasa izvedena je iz klase XmlNode, jer se i na dokument može gledati kao na jedan XML tag koji sadržava svoje podelemente
XmlElement	element u XML dokumentu
XmlAttribute	atribut elementa u XML dokumentu

Napravit ćemo dakle implementaciju kôda koji će s nekim XML dokumentom raditi kao s bazom podataka. Kako XML dokumenti služe prvenstveno za spremanje nekih podataka, ponekad je lakše iskoristiti njih nego stvarati i koristiti bazu. To će biti slučaj kad ne radite s pretjerano velikom količinom podataka ili jednostavno radite aplikaciju u kojoj ne želite koristiti bazu podataka (ili nemate mogućnost njenog korištenja).

Pretjerano korištenje baze podataka za svaku sitnicu može uvelike usporiti rad aplikacije. Radi li se o web-aplikaciji koja koristi jednu bazu podataka, velik broj istovremenih korisnika može otežati i potpuno onemogućiti rad s takvom aplikacijom koja se za sve oslanja na bazu. Stoga je preporučljivo manje podatke spremati u XML datotekama.

Radit ćemo s XML dokumentom koji će služiti za spremanje informacija o korisnicima aplikacije. Primjerice, uz svakog korisnika možete spremiti njegove postavke i druge informacije o njegovu načinu korištenja aplikacije, što vam može biti korisno.

Za početak, u tu ćemo konfiguracijsku datoteku spremati informacije o svim korisnicima koji su se prijavili za rad u aplikaciji. Pritom nećemo imati naglasak na sigurnosti – korisnici će upisati svoje ime, osnovne podatke, i to ćemo spremiti u XML datoteku. Kad se vrate aplikaciji, ponovno će upisati svoje ime, a program će automatski učitati njihove postavke.

Ovo je osnovna ideja rada aplikacije – mi nećemo napraviti pravu aplikaciju, već ćemo pokazati kako napisati kôd koji bi radio prethodne zadatke s XML datotekom. Evo i njenog oblika – dobro ga pogledajte, jer ćemo takvu datoteku stvarati i mijenjati:

```
<?xml version="1.0" encoding="utf-8" ?>
<Korisnici>
   <Korisnik Ime="Marko">
      <PunoIme>Marko Marić</PunoIme>
      <ZadnjaPrijava>10.5.2004 15:35:40</ZadnjaPrijava>
      <Postavke>
         <MaksimiziranProzor>True</MaksimiziranProzor>
         <Tema>Crna</Tema>
      </Postavke>
   </Korisnik>
   <Korisnik Ime="Petar">
      <PunoIme>Petar Petrović</PunoIme>
      <ZadnjaPrijava>10.5.2004 12:19:59</ZadnjaPrijava>
```

```
        <Postavke>
            <MaksimiziranProzor>False</MaksimiziranProzor>
            <Tema>Plava</Tema>
        </Postavke>
    </Korisnik>
</Korisnici>
```

Stvaranje XML dokumenta

Prvi korak u radu s XML dokumentom je njegovo stvaranje. Primjerice, ukoliko on ne postoji pri pokretanju aplikacije, stvorit će se novi XML dokument. Naredni kôd ne samo da stvara novi dokument već mu namješta XML deklaraciju (prvi <?xml ...?> *tag*) i glavni korijenski element ("Korisnici").

Pri radu s XML dokumentima ne zaboravite uključiti *namespace System.Xml* korištenjem ključne riječi *using*.

Slijedi metoda za stvaranje novog XML dokumenta.

```
private XmlDocument NoviXmlDokument()
{
    XmlDocument xml = new XmlDocument();

    XmlDeclaration dec = xml.CreateXmlDeclaration("1.0", "utf-8", null);
    xml.PrependChild(dec);
    XmlElement korijen = xml.CreateElement("Korisnici");
    xml.AppendChild(korijen);

    return xml;
}
```

Dakle, prvo se stvara novi objekt tipa *XmlDocument*. Potom se stvara XML deklaracija naredbom *CreateXmlDeclaration*, koja prima tri parametra – prvi predstavlja verziju korištenog XML standarda, druga *encoding* (koristimo univerzalan UTF-8 koji će bez problema prikazati naše znakove na

bilo kojem računalu), a treći prikazuje je li dokument samostalan (tzv. "standalone"), što nam za ovaj slučaj nije važno, pa smo proslijedili vrijednost *null*.

Korištenjem *PrependChild* naredbe dodajemo napravljenu deklaraciju XML dokumentu. *PrependChild* umeće element prije prvog elementa. Iako njega u tom trenutku još nismo stvorili, važno je znati da će ta naredba dodati element na sam početak zadane XML strukture.

Potom stvaramo korijenski element XML dokumenta. Za stvaranje elemenata u XML-u koristimo naredbu *CreateElement* kojoj prosljeđujemo naziv elementa. No samim stvaranjem elementa nismo ga dodali u strukturu XML dokumenta, već to moramo eksplicitno napraviti naredbom *AppendChild* koja dodaje novi element.

Prethodnu metodu možemo pozivati iz bilo kojeg dijela kôda:

```
XmlDocument mojXmlDokument = NoviXmlDokument();
```

Želimo li ispisati kompletan XML kôd dokumenta (primjerice, zbog provjere), iskoristili bismo *OuterXml* svojstvo XML dokumenta koje vraća njegov kompletan XML kôd, uključujući sve podelemente.

Spomenuli smo da se na cijeli XML dokument može gledati kao na jedan XML element sa svojim podelementima. To znači i da ćemo *OuterXml* svojstvo moći iskoristiti za ispis kompletnog XML kôda pojedinog elementa u strukturi, što će ispisati i sve njegove podelemente.

Primjerice, radite li prozorske aplikacije, kôd načinjenog XML dokumenta najjednostavnije biste ispisali:

```
MessageBox.Show(mojXmlDokument.OuterXml, "Provjera");
```

Rezultat poziva gornje metode bio bi tako sljedeći XML kôd:

```
<?xml version="1.0" encoding="utf-8"?>
<Korisnici/>
```

Kao što vidite, radi se o jednostavnom XML dokumentu koji sadrži deklaraciju i prazan korijenski element imena "Korisnici".

Dodavanje elemenata

Sad kad imate gotov i spreman XML dokument, možete u njega dodavati različite elemente. Evo kako biste dodali informacije za pojedinog korisnika – njegovo korisničko ime, puno ime, datum

Umjesto ručnog generiranja svakog elementa i njegova ubacivanja u XML strukturu novog dokumenta, mogli ste jednostavno učitati XML kôd korištenjem *loadXml* metode. Primjerice, sljedeći kôd rezultirao bi istim dokumentom kao i prethodni primjer:

```
XmlDocument xml = new XmlDocument();
xml.LoadXml("<?xml version=\"1.0\"
    encoding=\"utf-8\"?><Korisnici/>");
```

prijave te postavke – je li pri zadnjem korištenju aplikacije imao maksimiziran prozor i koju je temu odnosno sučelje aplikacije odabrao.

Za dodavanje novog korisnika rabi se metoda *DodajKorisnika*, a koristimo i dodatnu metodu *DodajTekstElement* koja prima nekoliko parametara – prvi određuje kojem XML dokumentu treba stvoriti novi element, drugi unutar kojeg postojećeg elementa treba dodati novi, a treći i četvrti određuju ime i vrijednost novog elementa. Primjerice, proslijedimo li za posljednja dva parametra vrijednosti "MaksimiziranProzor" i "True", stvorit će se novi element <MaksimiziranProzor>True</MaksimiziranProzor>.

```
private void DodajKorisnika(XmlDocument xml, string ime, string punoime, bool max-
prozor, string tema)
{
    XmlElement glavniElement = xml.DocumentElement;

    XmlElement Korisnik = xml.CreateElement("Korisnik");
    Korisnik.SetAttribute("Ime", ime);
    glavniElement.AppendChild(Korisnik);

    DodajTekstElement(xml, Korisnik, "PunoIme", punoime);
    DodajTekstElement(xml, Korisnik, "ZadnjaPrijava", DateTime.Now.ToString());

    XmlElement Postavke = xml.CreateElement("Postavke");
    Korisnik.AppendChild(Postavke);

    DodajTekstElement(xml, Postavke, "MaksimiziranProzor", maxprozor.ToString());
    DodajTekstElement(xml, Postavke, "Tema", tema);
}
```

```
private void DodajTekstElement(XmlDocument xml, XmlElement roditelj, string ime,
string vrijednost)
{
    XmlElement noviElement = xml.CreateElement(ime);
    XmlText sadrzaj = xml.CreateTextNode(vrijednost);
    noviElement.AppendChild(sadrzaj);

    roditelj.AppendChild(noviElement);
}
```

Dakle, metoda *DodajKorisnika* prima kao prvi parametar XML dokument unutar kojeg treba dodati strukturu za pojedinog korisnika. Drugi parametar predstavlja korisničko ime, treći puno ime, četvrti logičku vrijednost je li njegov prozor maksimiziran, a peti odabranu temu.

Kako ćemo strukturu *Korisnik* dodati ispod glavnog elementa, u varijablu imena *glavniElement* učitavamo referencu na glavni element u XML dokumentu. Ispod njega ćemo dodavati nove elemente.

To odmah radimo u sljedećim naredbama – stvaramo novi element naredbom *CreateElement* imena "Korisnik" i spremamo ga u varijablu *Korisnik*. Njemu dodajemo jedan atribut imena "Ime" i u njega upisujemo korisničko ime. To obavljamo uz pomoć naredbe *SetAttribute* koju pozivamo za neki element, u ovom slučaju novi element imena "Korisnik". Potom korištenjem već spomenute naredbe *AppendChild* dodajemo element "Korisnik" ispod glavnog elementa.

U ovom trenutku će struktura XML dokumenta imati sljedeći oblik:

```
<?xml version="1.0" encoding="utf-8"?>
<Korisnici>
   <Korisnik Ime="neko_ime"/>
</Korisnici>
```

Primijetite i da se tekstualni sadržaj nekog elementa smatra njegovim *djetetom*. U metodi *DodajTekstElement* stvaramo tekstualni sadržaj, a potom ga naredbom *AppendChild* dodajemo već postojećem elementu.

No sad u igru ulazi već opisana metoda *DodajTekstElement*. Njen je zadatak, dakle, da ispod nekog već postojećeg elementa u XML strukturi doda novi, tekstualni, oblika *<Ime>Vrijednost</Ime>*. Za dodavanje tekstualnog sadržaja pod neki element koristi se *XmlText* varijabla i naredba *CreateTextNode* koja dodaje sadržaj unutar nekog elementa.

Na kraju, u metodi *DodajTekstElement* dodajemo novi element s tekstualnim sadržajem ispod nekog postojećeg, koji je proslijeđen kao drugi parametar metodi.

I sad je sve jasno – u glavnoj metodi *DodajKorisnika* pozivamo metodu *DodajTekstElement* za svaki novi tekstualni element. Prva dva dodajemo ispod elementa "Korisnik", a nose imena "PunoIme" i "ZadnjaPrijava" (čiju vrijednost računamo iz trenutnog vremena, da dokument dobije na aktualnosti), a druga dva dodajemo ispod novog elementa "Postavke", a oni pak nose imena "MaksimiziranProzor" i "Tema".

Opisanu metodu *DodajTekstElement* možemo pozvati na standardan način i dodati korisnika postojećem XML dokumentu, primjerice:

```
DodajKorisnika(mojXmlDokument, "Marko", "Marko Marić", true, "Crna");
```

Rezultat poziva takve metode je sljedeći XML dokument:

```
<?xml version="1.0" encoding="utf-8"?>
<Korisnici>
   <Korisnik Ime="Marko">
      <PunoIme>Marko Marić</PunoIme>
      <ZadnjaPrijava>15.5.2004 17:53:38</ZadnjaPrijava>
      <Postavke>
         <MaksimiziranProzor>True</MaksimiziranProzor>
         <Tema>Crna</Tema>
      </Postavke>
   </Korisnik>
</Korisnici>
```

Kako ćete u radu s XML dokumentima najviše vremena provesti u radu s *XmlElement* objektima, u tablici 11-3 nalazi se popis njihovih korisnih svojstava.

Naravno, za dodavanje elemenata, njihovo brisanje i sve druge operacije ključne su metode *XmlElement* objekta, a neke važnije su navedene u tablici 11-4.

Tablica 11-3:
Neka korisna svojstva XmlElement objekata

Svojstvo	Opis
Attributes	vraća kolekciju tipa *XmlAttributeCollection* koja sadrži sve atribute elementa
ChildNodes	vraća sve podelemente
FirstChild	vraća prvi podelement
HasAttributes	vraća logičku vrijednost, ima li element neke atribute
HasChildNodes	vraća logičku vrijednost, ima li element *djecu* (odnosno sadrži li podelemente)
InnerText	tekstualni sadržaj elementa
InnerXml XML	sadržaj elementa (dakle, samo sva njegova djeca, bez XML kôda samog elementa)
LastChild	vraća posljednji podelement Name vraća ime elementa
NextSibling	vraća sljedeći element u hijerarhiji (tzv. *bratski element*, jer se nalaze na istoj razini hijerarhije)
OuterXml	vanjski XML sadržaj elementa (uključuje XML kôd samog elementa i sve njegove djece)
ParentNode	vraća nadelement trenutnog (tzv. *roditelj*)
PreviousSibling	vraća prethodni element u hijerarhiji (tzv. *bratski element*, jer se nalaze na istoj razini hijerarhije)

Proširenja DOM-a

Microsoft je ipak proširio standardne definirane mogućnosti DOM API-ja u klasi *XmlDocument*. Tako vam na raspolaganju stoji svojstvo *InnerText*, pomoću kojeg možete definirati tekst unutar nekog elementa. Primjerice, metodu *DodajTekstElement* mogli ste napisati na sljedeći način:

```
private void
DodajTekstElement(XmlDocument xml,
XmlElement roditelj, string ime,
string vrijednost)
{
    XmlElement noviElement =
        xml.CreateElement(ime);
    noviElement.InnerText =
        vrijednost;
    roditelj.AppendChild(noviElement);
}
```

Kao što vidite, radi se o pojednostavljenju i svojevrsnom unaprjeđenju mogućnosti rada s XML dokumentima. Korištenjem *InnerText* svojstva direktno smo upisali tekstualnu vrijednost nekom elementu, a nismo morali stvarati novi element tipa XmlText naredbom *CreateTextNode* te ga s *AppendChild* dodavati.

Tablica 11-4:
Neke korisne metode XmlElement objekata

Metoda	Opis
AppendChild	dodaje novi element unutar postojećeg (novi element se dodaje na kraj odnosno poslije svih postojećih podelemenata)
GetAttribute	vraća vrijednost zadanog atributa
GetElementsByTagName	vraća sve podelemente zadanog imena
HasAttribute	vraća logičku vrijednost, ima li element zadani atribut
PrependChild	dodaje novi element unutar postojećeg (novi element se dodaje na, odnosno prije svih postojećih podelemenata)
RemoveAttribute	briše zadani atribut elementa
RemoveChild	briše zadani podelement
ReplaceChild	zamjenjuje zadani podelement novim elementom
SelectNodes	vraća listu elemenata dohvaćenu XPath upitom
SelectSingleNode	vraća prvi element dohvaćen XPath upitom
SetAttribute	Postavlja vrijednost atributu

Traženje elemenata

Kao što u svom programu želite upisivati podatke o novim korisnicima, tako ih vrlo vjerojatno želite i mijenjati. Iako je naredne primjere moguće izraditi i uz pomoć XPatha (što bi bilo brže i efikasnije), to ćemo ipak ostaviti za kasnije, a sad ćemo se pozabaviti korištenjem DOM-a za rad s XML dokumentom.

Za početak će na trebati metoda koja će pronaći strukturu korisnika odnosno odgovarajući element "Korisnik", koji se nalazi ispod elementa "Korisnici". Kako smo na početku programa načinili pretpostavku da korisničko ime jedinstveno određuje nekog korisnika (npr. ne mogu postojati dva korisnika s korisničkim imenom "Marko"), tražit ćemo element "Korisnik" koji ima zadano korisničko ime.

Evo kako bi izgledala ta metoda:

```
private XmlElement PronadjiKorisnika(XmlDocument xml, string ime)
{
```

```
    XmlElement pronadjeniKorisnik = null;
    XmlElement glavniElement = xml.DocumentElement;

    XmlNodeList listaKorisnika = glavniElement.GetElementsByTagName("Korisnik");
    foreach (XmlNode korisnik in listaKorisnika)
    {
        if (korisnik is XmlElement)
        {
            XmlElement elementKorisnik = (XmlElement) korisnik;
            if (ime == elementKorisnik.GetAttribute("Ime"))
            {
                pronadjeniKorisnik = elementKorisnik;
                break;
            }
        }
    }
    return pronadjeniKorisnik;
}
```

Iako smo mogli koristiti XPath, cilj je bio pokazati korištenje *XmlNodeList* odnosno liste *XmlNode* objekata u strukturi XML dokumenta.

Dakle, metoda *PronadjiKorisnika* pretražuje XML dokument koji joj je proslijeđen kao prvi parametar u potrazi za korisnikom s korisničkim imenom koje je zadano kao drugi parametar. Ona vraća pronađeni element.

U njoj najprije određujemo varijablu *pronadjeniKorisnik* u koju će biti spremljena referenca na element u XML strukturi i postavljamo je na *null* jer ga još nismo našli. U varijablu *glavniElement* učitavamo korijenski element XML strukture.

Nad njim izvršavamo naredbu *GetElementsByTagName* koja vraća listu svih čvorova sa zadanim imenom (u našem slučaju "Korisnik"). Podsjetimo, *XmlNode* objekt može biti običan element, može biti neki atribut ili bilo koji drugi dio XML dokumenta. Zatim u *foreach*-petlji prolazi kroz sve dohvaćene zapise sa zadanim imenom. Kako lista dohvaćenih zapisa sadržava *XmlNode* objekte, tako ih sve u petlji spremamo u varijablu *korisnik* tog tipa.

No kako tražimo samo obične XML elemente (dakle, ne bi nam odgovarali, primjerice, atributi s tim imenom), u samoj petlji radimo provjeru je li objekt *korisnik* zapravo tipa *XmlElement*. Ukoliko je to XML element, eksplicitno ga pretvaramo iz *XmlNode* tipa u očekivani *XmlElement* jer ćemo tako moći pristupati njegovim atributima (sjetimo se petog poglavlja – radi se o *boxingu* i *unboxingu*).

Potom vršimo usporedbu – ukoliko parametar *ime* odgovara atributu "ime" dohvaćenog elementa, pronašli smo ono što smo tražili. U varijablu *pronadjeniKorisnik* spremamo referencu na pronađeni element te prekidamo petlju i vraćamo varijablu kao rezultat metode.

U slučaju da ne pronađemo korisnika sa zadanim korisničkim imenom, metoda bi vratila null vrijednost jer je to početna vrijednost varijable pronadjeniKorisnik.

Tu metodu možemo pozvati iz glavnog dijela programa. Evo kompletnog izvornog kôda:

```
XmlDocument mojXmlDokument = NoviXmlDokument();
DodajKorisnika(mojXmlDokument, "Marko", "Marko Marić", true, "Crna");

XmlElement k = PronadjiKorisnika(mojXmlDokument, "Marko");
if (k != null) MessageBox.Show(k.OuterXml, "Provjera");
```

U dokumentu *mojXmlDokument* tražimo korisnika s imenom "Marko". Ako on ne bude pronađen (što ipak neće biti slučaj jer ga dodajemo jednu naredbu ranije), poruka se neće ispisati – objekt *k* u kojem je spremljen rezultat poziva bit će *null* i neće se izvršiti naredba *MessageBox.Show*. Naravno, ako se pronađe odgovarajući korisnik, ispisuje se samo XML kôd unutar pronađenog elementa imena "Korisnici". Evo kako taj kôd izgleda:

```
<Korisnik Ime="Marko">
   <PunoIme>Marko Marić</PunoIme>
   <ZadnjaPrijava>15.5.2004 17:53:38</ZadnjaPrijava>
   <Postavke>
      <MaksimiziranProzor>True</MaksimiziranProzor>
      <Tema>Crna</Tema>
   </Postavke>
</Korisnik>
```

Uočite da smo ispisali *OuterXml* svojstvo nekog elementa i tako dobili kompletan njegov XML kôd, uključujući i podelemente.

Mijenjanje elemenata

Nakon što dohvatite neki element prethodno opisanom metodom, vrlo lako možete mijenjati njegove podelemente ili atribute. Evo kako bi izgledala metoda koja je izmijenila puno ime korisnika i

datum njegove prije, tj. osvježila ga na ažurniju vrijednost (zbog jednostavnosti nećemo mijenjati ostale elemente, no princip je isti). I ovaj put ćemo, da pokažemo mogućnosti rada s DOM-om, koristiti što više različitih metoda i svojstava objekata za rad s XML-om.

```
private void IzmijeniKorisnika(XmlDocument xml, string ime, string novoPunoIme)
{
   XmlElement korisnik = PronadjiKorisnika(xml, ime);
   if (korisnik != null)
   {
      XmlNodeList listaElemenata = korisnik.ChildNodes;
      for (int i = 0; i < listaElemenata.Count; i++)
      {
         if (listaElemenata.Item(i) is XmlElement)
         {
            XmlElement element = (XmlElement) listaElemenata.Item(i);
            if (element.Name == "PunoIme")
            {
               XmlText noviSadrzaj = xml.CreateTextNode(novoPunoIme);
               element.ReplaceChild(noviSadrzaj, element.FirstChild);
               break;
            }
         }
      }
   }
}
```

Dakle, metodi prosljeđujemo XML dokument u kojem treba načiniti izmjenu, korisničko ime korisnika te njegovo novo puno ime koje treba upisati umjesto postojećeg. Prvo koristimo načinjenu metodu *PronadjiKorisnika,* koja vraća element korisnika sa zadanim korisničkim imenom.

U varijablu *listaElemenata* učitavamo listu svih podelemenata pronađenog korisnika – lista svih podelemenata dostupna nam je preko svojstva *ChildNodes*. Potom koristimo običnu *for*-petlju za kretanje kroz sve dohvaćene podelemente (mogli smo koristiti i *foreach*-petlju, no čisto da pokažemo nešto novog kôda).

Ukoliko je neki podelement tipa *XmlElement* odnosno ako se radi o pravom elementu, eksplicitno ga pretvaramo i spremamo u varijablu *Element*. Zatim provjeravamo njeno ime pomoću svojstva *Name* – ukoliko je ono "PunoIme", znači da smo pronašli element u kojem je spremljeno puno ime korisnika.

Tad stvaramo novi tekstualni element korištenjem *CreateTextNode* s varijablom koja sadrži novo puno ime. Korištenjem *ReplaceChild* metode zamjenjujemo jedan podelement drugim – kao prvi

parametar navodimo novi element, a kao drugi element koji želimo zamijeniti. Kako je tekstualni sadržaj jedini podelement elementa "PunoIme", koristimo *FirstChild* svojstvo i na njegovo mjesto stavljamo novi sadržaj.

I u ovom ste slučaju mogli iskoristiti *InnerText* svojstvo za zapis tekstualnog sadržaja elementa. Izmijenjeni dio kôda u tom bi slučaju imao sljedeći oblik:

```
if (element.Name == "PunoIme")
{
    element.InnerText = novoPunoIme;
    break;
}
```

Brisanje elemenata

Važna stavka u radu s XML dokumentima je i brisanje elemenata u XML strukturi. Sve ćemo opet objasniti na primjeru brisanja korisnika. Slijedi metoda kojoj biste proslijedili korisničko ime korisnika, a ona bi pronađenu strukturu izbrisala:

```
private void BrisiKorisnika(XmlDocument xml, string ime)
{
    XmlElement korisnik = PronadjiKorisnika(xml, ime);
    if (korisnik != null)
    {
        XmlNode nadElement = korisnik.ParentNode;
        nadElement.RemoveChild(korisnik);
    }
}
```

Metoda je jednostavna – prvo pronalazi korisnika, zatim u varijablu *nadElement* učitava njegova *roditelja* korištenjem *ParentNode* svojstva. Potom korištenjem *RemoveChild* metode nad objektom nadelementa, prosljeđujući joj referencu na element koji treba izbrisati, briše cijelu strukturu elementa "Korisnik", što uključuje i sve njegove podelemente.

Učitavanje i spremanje XML dokumenata

Naravno, sve ove naredbe i način korištenja neće vam previše vrijediti ukoliko ne spremite sve promjene nad XML dokumentom u neku datoteku. Tu datoteku kasnije možete opet učitati i nastaviti gdje ste stali – tako ona dobiva funkciju svojevrsne konfiguracijske datoteke jer u njoj možete spremati različite postavke koje će biti čuvane na disku računala.

Za učitavanje i spremanje datoteka koriste se *Load* i *Save* metode XML dokumenta koje za parametar primaju naziv datoteke XML dokumenta. Primjerice, želite li na početku rada aplikacije učitati neku XML datoteku, napišite sljedeći kôd:

```
XmlDocument mojXmlDokument = NoviXmlDokument();
mojXmlDokument.Load("korisnici.xml");
```

Prethodni bi primjer tako učitao u XML dokument sadržaj datoteke "korisnici.xml" koja se nalazi u istom direktoriju kao i sama aplikacija. Naravno, možete navesti i punu putanju do XML datoteke, primjerice "C:\Programi\Aplikacija\korisnici.xml".

Osim učitavanja datoteka s lokalnog diska, imate mogućnost i učitavanja datoteka s Interneta. Primjerice, ako biste željeli u svom programu učitati XML datoteku smještenu na nekom poslužitelju na Internetu, samo biste trebali napisati njenu punu adresu:

```
mojXmlDokument.Load("http://www.server.com/aplikacija/korisnici.xml");
```

Pri dohvaćanju XML datoteka s Interneta, naravno, preporučljivo je ugraditi neki mehanizam obrade iznimki – što ako datoteka na tom serveru ne postoji ili što ako je server trenutno nedostupan? To su slučajevi za koje se ipak trebate pripremiti u svojem programu.

XML datoteke, se spremaju slično tome kako se učitavaju. Kao parametar *Save* metodi samo navedite naziv datoteke u koju želite spremiti XML strukturu. Takvu ćete akciju najčešće obavljati po završetku rada s programom, kako biste sačuvali sve izmjene.

```
mojXmlDokument.Save("korisnici.xml");
```

Korištenje XML datoteka za spremanje konfiguracijskih postavki aplikacije je dobra ideja, a to potvrđuje i ASP.NET koji u web.config datoteci sprema konfiguracijske postavke svake aplikacije.

Pisanje XML dokumenata

Želite li stvoriti novi XML dokument i zapisati ga u neku datoteku, možete iskoristiti klasu *XmlTextWriter*. Ona služi za slijedno zapisivanje podataka i mnogo je brža nego da u memoriji stvarate novu strukturu XML dokumenata stvarajući poseban objekt za svaki element. U tablici 11-5 navedene su njene osnovne metode.

Tablica 11-5:
Osnovne metode za korištenje klase XmlTextWriter

Metoda	Opis
WriteAttributes	zapisuje sve atribute pronađene na trenutnoj poziciji u *XmlReaderu*
WriteAttributeString	zapisuje atribut s određenom vrijednošću
WriteCData	zapisuje <![CDATA[…]]> blok sa zadanim tekstom
WriteComment	zapisuje komentar (<!— … —>) sa zadanim tekstom
WriteElementString	zapisuje element
WriteStartAttribute, WriteEndAttribute	otvara novi atribut odnosno zatvara atribut
WriteStartDocument, WriteEndDocument	otvara strukturu XML dokumenta odnosno zatvara strukturu dokumenta
WriteStartElement, WriteEndElement	otvara pisanje novog elementa odnosno zatvara pisanje elementa
WriteRaw	zapisuje čisti XML kôd napisan ručno

Da bismo pokazali način korištenja klase *XmlTextWriter*, napravit ćemo metodu koja će zapisati strukturu našeg oglednog XML dokumenta s jednim korisnikom u neku datoteku.

```
private void ZapisiXmlDokument(string datoteka)
{
    XmlTextWriter xml = new XmlTextWriter(datoteka, System.Text.Encoding.UTF8);
    xml.Formatting = Formatting.Indented;

    // Otvaramo strukturu XML dokumenta
    xml.WriteStartDocument();
```

```
        // Zapisujemo komentar
        xml.WriteComment("Automatski generirani XML dokument");

        // Otvaramo element "Korisnici"
        xml.WriteStartElement("Korisnici");

        // Otvaramo element "Korisnik" i zapisujemo mu atribut
        xml.WriteStartElement("Korisnik");
        xml.WriteAttributeString("Ime", "Marko");

        // Zapisujemo element "PunoIme"
        xml.WriteStartElement("PunoIme");
        xml.WriteString("Marko Marić");
        xml.WriteEndElement();

        // Zapisujemo element "ZadnjaPrijava"
        xml.WriteStartElement("ZadnjaPrijava");
        xml.WriteString(DateTime.Now.ToString());
        xml.WriteEndElement();

        // Zatvaramo element "Korisnik"
        xml.WriteEndElement();

        // Zatvaramo element "Korisnici"
        xml.WriteEndElement();

        // Zatvaramo strukturu XML dokumenta
        xml.WriteEndDocument();

        xml.Close();
    }
```

Kôd metode je vrlo jasan – krećemo se naredbu po naredbu i upisujemo elemente u XML dokument. Ključno je otvaranje dokumenta – kao parametar konstruktoru klase *XmlTextWriter* prosljeđujemo naziv datoteke (koji metoda prima kao parametar) i *encoding* koji se nalazi u pobrojanom nizu *System.Text.Encoding* pa tako možete izabrati i ASCII *encoding* izlazne datoteke.

Postavljanjem svojstva *Formatting* određujemo kako će biti oblikovana ciljna datoteka – u pobrojanom nizu *Formatting* odabrali smo vrijednost *Indented*, što znači da će elementi u XML dokumentu biti uvučeni ovisno o njihovoj razini, što će rezultirati pregledno oblikovanim dokumentom.

Metodu iz primjera pozivamo na standardni način iz glavnog programa:

```
ZapisiXmlDokument("korisnici.xml");
```

Rezultat njenog poziva bit će nova datoteka sa zadanim imenom i upisanim XML sadržajem.

Kasnije je sve jasno iz komentara – prvo otvaramo strukturu XML dokumenta, zapisujemo komentar, zatim otvaramo element "Korisnici" i u njemu upisujemo novi element "Korisnik" s jednim atributom. Nastavljamo na isti način dalje – stvaramo elemente "PunoIme" i "ZadnjaPrijava", a potom zatvaramo još nezatvorene strukture elemenata "Korisnik", "Korisnici" te cijele strukture XML dokumenta. Klasu *XmlTextWriter* zatvaramo s *Close()*, a nova datoteka na disku imat će sljedeći sadržaj:

```
<?xml version="1.0" encoding="utf-8"?>
<!–Automatski generirani XML dokument–>
<Korisnici>
   <Korisnik Ime="Marko">
      <PunoIme>Marko Marić</PunoIme>
      <ZadnjaPrijava>21.5.2004 11:53:11</ZadnjaPrijava>
   </Korisnik>
</Korisnici>
```

Korištenje klase *XmlTextWriter* preporučuje se kad želite nabrzinu zapisati neki sadržaj i stvoriti novu XML datoteku. Kao što je rečeno, takav pristup je mnogo brži nego stvaranjem svakog posebnog elementa, pozivanjem *AppendChild* metode i slično. Uvijek tako stvorenu datoteku možete i učitati u objekt tipa *XmlDocument* korištenjem metode *Load*.

XPath

Zahvaljujući SQL jeziku za pristup podacima, baze podataka definitivno su otvorile svoja vrata svim znatiželjnicima. Korištenjem lako razumljivog SQL jezika, koji je zapravo industrijski standard koji sve baze više-manje doslovno implementiraju, korisnici imaju mogućnost dohvaćanja i obrade svih podataka spremljenih u bazi.

Kako se može reći da i XML format služi prvenstveno za pohranu nekih podataka, pojava standarda za pristup tim podacima bio je neupitan. Zahvaljujući W3C-u, on se 1999. godine pojavio u obliku XPatha i predstavlja ključni element u bilo kakvom radu s XML dokumentima. Kad ga već uspoređujemo sa SQL-om, budimo potpuno precizni – dok SQL omogućava i mijenjanje i dodavanje podataka, XPath služi isključivo za dohvaćanje dijelova XML dokumenata.

Osnove XPatha

XPath za pristup dijelovima XML dokumenata koristi sintaksu vrlo sličnu onoj na koju ste navikli pri radu s datotekama ili web-stranicama, primjerice direktorij/direktorij/datoteka. Naravno, u kontekstu XML dokumenata, ne radi se o direktorijima i datotekama, već o elementima odnosno *nodeovima* u hijerarhiji.

Da bismo potpuno objasnili mogućnosti XPatha, trebat ćemo i XML dokument koji ćemo koristiti u primjerima. Radi se o dokumentu čiju ste strukturu upoznali već u prethodnim primjerima, no ponovit ćemo ga još jednom da lakše možete pratiti XPath upite:

```
<?xml version="1.0" encoding="utf-8" ?>
<Korisnici>
   <Korisnik Ime="Marko">
      <PunoIme>Marko Marić</PunoIme>
      <ZadnjaPrijava>10.5.2004 15:35:40</ZadnjaPrijava>
      <Postavke>
         <MaksimiziranProzor>True</MaksimiziranProzor>
         <Tema>Crna</Tema>
      </Postavke>
   </Korisnik>
   <Korisnik Ime="Petar">
      <PunoIme>Petar Petrović</PunoIme>
      <ZadnjaPrijava>10.5.2004 12:19:59</ZadnjaPrijava>
      <Postavke>
         <MaksimiziranProzor>False</MaksimiziranProzor>
         <Tema>Plava</Tema>
      </Postavke>
   </Korisnik>
</Korisnici>
```

Kretanje po strukturi

Dakle, cilj XPath upita je odabrati jedan ili više elemenata ili njihovih atributa. Doista je mnogo kriterija po kojima možete vršiti selekciju, no krenimo s najjednostavnijim. Slijede tri upita:

```
/Korisnici
/Korisnici/Korisnik
/Korisnici/Korisnik/PunoIme
```

Upiti se rade tako da se napiše struktura koja se želi označiti, a XPath će pronaći sve elemente koji odgovaraju zadanoj strukturi. Prvi upit će, kao što možete i očekivati, označiti glavni korijenski element XML dokumenta, imena "Korisnici". Drugi će pak upit označiti sve elemente imena "Korisnik" koji se nalaze ispod elementa "Korisnici". U našem oglednom XML dokumentu postoje dva takva elementa. Slična je situacija i s trećim upitom koji označava sve elemente "PunoIme" koji se nalaze ispod jednog od elemenata "Korisnik", koji se pak nalaze ispod elementa "Korisnici". Tim trećim upitom dohvatili biste puna imena svih korisnika iz XML dokumenta.

Uočite da svi ti upiti počinju sa *slash* znakom ("/") – on označava da se radi o apsolutnom putu, tj. onom koji počinje na vrhu hijerarhijske XML strukture. No stavite li na početak upita dva *slash* znaka, pretraživat ćete cijelu XML hijerarhiju za traženim elementom.

```
//Tema
```

Gornji će primjer tako za rezultat dobiti sve elemente "Tema" koji se nalaze bilo gdje u hijerarhijskoj strukturi.

Osim određivanja točnog puta za odabir elemenata, možete iskoristit jedan ili više *wildcard* znakova (*) za označavanje svih mogućnosti, kao što je vidljivo u narednim primjerima:

```
/Korisnici/Korisnik/*
/Korisnici/*/PunoIme
/*/*/PunoIme
```

U prvom primjeru odabrani su svi elementi koji se nalaze ispod hijerarhije /Korisnici/Korisnik, a to su svi elementi "PunoIme", "ZadnjaPrijava", "Postavke" u dokumentu. Drugi primjer pak označava sve elemente "PunoIme", koji se nalaze u drugoj razini ispod elementa "Korisnici" (primjerice, označio bi i elemente /Korisnici/nesto_treće/PunoIme da postoje). Treći primjer pak označava sve elemente imena "PunoIme" koji imaju dva pretka bilo kojeg imena u hijerarhijskoj strukturi.

Napišete li "//*", označit ćete sve elemente u dokumentu jer, podsjetimo, dva *slash* znaka označavaju da se pretražuju svi elementi bilo gdje unutar strukture, a zvjezdica da upitu odgovaraju elementi bilo kojeg imena.

U prethodnim smo primjerima označavali sve elemente koji bi odgovarali zadanom upitu. Ponekad ćete željeti odabrati samo neke od njih. Pogledajte sljedeće primjere:

```
/Korisnici/Korisnik[1]
/Korisnici/Korisnik[last()]
/Korisnici/Korisnik[PunoIme='Marko Marić']
```

Unutar uglatih zagrada možete napisati dodatne uvjete. Dakle, prvi upit će označiti prvi element "Korisnik" koji se nalazi ispod elementa "Korisnici". Drugi upit će pak označiti zadnji element "Korisnik" koji se nalazi ispod elementa "Korisnici", no primijetite da ne postoji funkcija poput *first()* koja bi vratila prvi element – želite li to postići, samo napišite [1], kao u prvom primjeru. Treći primjer pak označava element "Korisnik" koji sadrži element "PunoIme", koji pak ima vrijednost "Marko Marić". Rezultat tog upita bio bi prvi element "Korisnik".

Atributi elemenata

Osim po imenu elemenata, možete pretraživati i po njihovim atributima. Krenimo opet s primjerima:

```
//@Ime
//Korisnik[@Ime='Marko']
//Korisnik[@*]
```

U XML-u na atribute elemenata se gleda baš kao i na elemente – oni predstavljaju zasebne čvorove i također se mogu označiti i iz njih izvući podaci. Prvi primjer tako označava sve atribute "Ime" u XML dokumentu te tako dobivate sva korisnička imena u dokumentu, u konkretnom primjeru dva – "Marko" i "Petar".

Drugi primjer označava sve elemente "Korisnik" kojima je atribut "Ime" jednak "Marko" odnosno korisnika s korisničkim imenom "Marko". Treći pak primjer označava sve elemente "Korisnik" koji u sebi sadržavaju bilo koji atribut. Uočite da se u svim primjerima koriste dva *slash* znaka koji određuju pretraživanje svuda unutar XML hijerarhije.

Usporedbe i odnosi

Osim dosad prikazanog operatora jednakosti ("="), u XPathu možete koristiti i niz drugih operatora. Želite li tako provjeriti nejednakost, iskoristit ćete "!=", tj. *not equal* operator. Naravno, dostupne su vam i druge usporedbe, primjerice *manje-od* ("<"), *manje-ili-jednako* ("<="), *veće-od* (">") i *veće-ili-jednako* (">=").

Nad vrijednostima (primjerice, atributa) u XML dokumentu možete obavljati i aritmetičke operacije – zbrajanje ("+"), oduzimanje ("-"), množenje ("*"), dijeljenje ("div") i računanje ostatka dijeljenja ili modul ("mod"). Ako vam to nije dovoljno, možete koristiti i logičke operatore AND ("and") i OR ("or").

Osim što uz pomoć XPatha možete označivati elemente u odnosu na cijelu XML hijerarhiju (tj. apsolutno), to možete raditi i relativno, tj. u odnosu na trenutno označeni element. Pri korištenju XML

dokumenata u svom kôdu, radit ćete s različitim elementima unutar hijerarhije. Nakon što učitate XML dokument, bit ćete pozicionirani na glavni element u hijerarhiji i u odnosu na njega ćete obavljati pretraživanja. No vi vrlo lako možete označiti neki drugi element i zatim nad njim obavljati XPath upite koji će označavati sve njegove podelemente, baš kao da je on glavni element u hijerarhiji.

U tablici 11-6 možete vidjeti nekoliko ključnih riječi koje vam omogućavaju relativno označavanje elementa, tj. u ovisnosti o njihovu odnosu.

Tablica 11-6:
Nekoliko ključnih riječi koje će vam koristiti kad budete označavali XML elemente u odnosu na trenutno označeni

Metoda	Opis
ancestor::	svi prethodnici trenutnog elementa – njegov nadelement, zatim njegov nadelement pa tako sve do glavnog elementa u hijerarhiji
ancestor-or-self::	trenutni element i svi njegovi prethodnici
attribute::	svi atributi trenutnog elementa
child::	djeca odnosno podelementi trenutnog elementa
descendant::	suprotno od *ancestors* – svi podelementi trenutnog elementa (djeca trenutnog, zatim njihova djeca itd.)
descendant-or-self::	trenutni element i svi njegovi *nasljednici*
following::	svi sljedeći elementi u hijerarhiji (ne uključuje *descendant* elemente od trenutnog)
following-sibling::	svi sljedeći elementi koji se nalaze ispod istog elementa kao i trenutni (*braća*)
parent::	nadelement (*roditelj*) trenutnog elementa preceding:: svi prethodni elementi u hijerarhiji
preceding-sibling::	svi prethodni elementi koji se nalaze ispod istog elementa kao i trenutno (*braća*)
self::	trenutni element

Slijedi nekoliko primjera koji će zorno prikazati njihovu uporabu:

```
/Korisnici/Korisnik[2]/preceding::*
/Korisnici/child::Korisnik[position()<3]
child::Korisnik[position()<3]
```

Prvi XPath upit kreće od drugog elementa "Korisnik" te zatim označava sve njegove prethodnike u XML hijerarhiji, a to su prvi element "Korisnik" i sva njegova djeca. Za objašnjavanje drugog i trećeg upita potrebno je pojasniti čemu služi funkcija *position()* – ona vraća poziciju nekog elementa unutar nadelementa. Tako će drugi upit vratiti sve podelemente od "Korisnici" imena "Korisnik", i to samo prva dva, jer su njihove pozicije manje od 3 (naravno, u našem dokumentu postoje samo dva, no u slučaju da ih ima više, taj upit bi vratio samo prva dva).

Treći upit obavlja istu stvar, no u ovisnosti o trenutno odabranom elementu – ukoliko ga izvršite nad elementom "Korisnici", dobit ćete isti rezultat kao i u prethodnom primjeru.

Kratice

Ne trebate uvijek koristiti ključne riječi poput *child::* ili *parent::*. Za njih postoje već objašnjene kratice. Primjerice, napišete li "child::tekst", to je isto kao da ste napisali i samo "tekst" jer će XPath uvijek krenuti s pretraživanjem podelemenata.

Isto tako, napišete li samo ".", to je isto kao da ste napisali i *self::* odnosno označili trenutan element. Primjerice, "./tekst" je identično "self::node()/child::tekst". Slično vrijedi i za *parent::* koji je jednak ".." pa su tako sljedeća dva izraza jednaka: "../tekst" i "parent::node()/child::tekst".

I na kraju, pretraživanje svih XML elemenata u hijerarhiji s dva *slash* znaka moguće je izvesti i s *descendant-or-self::*, primjerice "//tekst" je isto što i "/descendant-or-self::node()/child::cd".

Rad s XPathom

U .NET-u s XPathom možete raditi preko klase *XmlNode* nekog objekta *XmlDocument* i preko klase *XPathNavigator*. Kao što ste mogli vidjeti u tablici 11-4, na raspolaganju vam stoje dvije metode – *SelectNodes* i *SelectSingleNode* – koje koriste XPath upite za pretraživanje dokumenta.

Primjerice, želite li ispisati sve vrijednosti elemenata "PunoIme" odnosno dohvatiti imena svih korisnika iz XML dokumenta, možete iskoristiti sljedeći kôd (pretpostavka je da i dalje koristite XML dokument kao u prvim primjerima te da je on učitan u varijablu *mojXmlDokument*):

```
XmlNodeList imena = mojXmlDokument.SelectNodes("//PunoIme");
foreach (XmlNode ime in imena)
{
```

```
    MessageBox.Show(ime.InnerText, "Ime");
}
```

Prethodni kôd je već jasan – dohvaćate listu elemenata, baš kao što ste radili u prethodnim primjerima s *ChildNodes* ili *GetElementsByTagName*. Ta se lista sprema u objekt tipa *XmlNodeList*, a zatim koristi *foreach*-petlja za kretanje kroz sve dohvaćene elemente. Potom se samo ispisuje njihov sadržaj u prozoru za poruke – ta ste imena mogli učitati u neki padajući izbornik ili bilo koju drugu kontrolu.

Na sličan način radi i *SelectSingleNode* metoda koja, kao što je rečeno, vraća samo jedan element. Evo kako bi izgledao primjer u kojem dohvaćamo puno ime korisnika s korisničkim imenom "Marko".

```
XmlNode ime =
    mojXmlDokument.SelectSingleNode("/Korisnici/Korisnik[@Ime='Marko']/PunoIme");

MessageBox.Show(ime.InnerText, "Ime");
```

Ako XPath upit proslijeđen *SelectSingleNode* metodi vrati više elemenata, svi će se zanemariti osim prvog elementa, koji će biti dohvaćen kao rezultat poziva.

No kompletan API u .NET-u za rad s XPath upitima je izgrađen oko klase *XPathNavigator*. Pomoću nje i klase *XPathDocument* ćemo u narednom primjeru učitati listu svih imena korisnika.

U klasu *XPathDocument* učitat ćemo XML dokument nad kojim ćemo zadavati upite. Korištenjem klase *XPathNavigator* zadat ćemo upit koji će vratiti određeni set rezultata, a uz pomoć klase *XPathNodeIterator* ćemo se kretati kroz dohvaćene elemente. Pogledajte primjer:

```
XPathDocument xml = new XPathDocument("korisnici.xml");
XPathNavigator nav = xml.CreateNavigator();
XPathNodeIterator iter = nav.Select("/Korisnici/Korisnik/PunoIme");

while (iter.MoveNext())
{
    MessageBox.Show(iter.Current.Value, "Ime");
}
```

Korištenjem klasa za rad s XPathom ubrzavate svoje aplikacije i preporučljivo ih je koristiti jer se one ionako koriste u pozadini poziva *SelectNode* naredbe. Dakle, u primjeru stvaramo novi *XPathDocument* objekt, a iz njega pozivom metode *CreateNavigator* stvaramo objekt tipa *XPathNavigator*. On će nam služiti za zadavanje XPath upita.

Objekt tipa *XPathNodeIterator* služi nam za kretanje kroz dohvaćene elemente – u njega učitavamo listu elemenata korištenjem objekta *XPathNavigator* i metode *Select* u kojoj navodimo XPath upit.

Korištenjem *while*-petlje krećemo se kroz sve elemente, a trenutnom elementu pristupamo svojstvom *Current*. Na kraju ispisujemo vrijednost svih elemenata korištenjem *Value* svojstva.

Dohvatite li XPath upitom vrijednost nekog atributa, njega također možete ispisati *Value* svojstvom – ono je predviđeno da vrati vrijednost bilo kojeg dohvaćenog elementa, bio to običan XML element ili neki atribut.

Vrlo često ćete i XPath upite koristiti da biste dohvatili neke elemente i kasnije s njima radili – mijenjali ih ili čak brisali. No ukoliko im pristupate preko *XPathNodeIterator* kolekcije, oni su *read-only*, što znači da samo možete pročitati njihovu vrijednost, a ne možete je mijenjati.

Naredni primjer stvara *XPathNavigator* iz običnog XML dokumenta tipa *XmlDocument* i pretvara dohvaćeni element u tip *XmlNode* – time dobivate punu kontrolu nad elementom i nad njim možete raditi sve preko DOM-a. Iskoristit ćemo to za promjenu svih korisničkih imena – umjesto običnih imena, dat ćemo im brojeve koji odgovaraju rednom broju elementa "Korisnik" u XML strukturi dokumenta.

```
XmlDocument mojXmlDokument = NoviXmlDokument();
mojXmlDokument.Load("korisnici.xml");

XPathNavigator nav = mojXmlDokument.CreateNavigator();
XPathNodeIterator iter = nav.Select("/Korisnici/Korisnik");

XmlNode element;

while (iter.MoveNext())
{
   element = ((IHasXmlNode) iter.Current).GetNode();
   element.Attributes["Ime"].Value = iter.CurrentPosition.ToString();
}
```

Dakle, objekt tipa *XPathNavigator* stvaramo iz objekta *XmlDocument* tipa pozivom standardne metode *CreateNavigator*, a zatim u *XPathNodeIterator* učitavamo sve elemente koji odgovaraju zadanom XPath upitu. U gornjem slučaju to su svi elementi "Korisnik".

Za pristup pojedinom elementu koristit ćemo varijablu *element* tipa *XmlNode*. U petlji kojom se krećemo kroz listu dohvaćenih elemenata XPath upitom, u varijablu *element* učitavamo pojedini element. To obavljamo konverzijom trenutnog elementa u tip "IHasXmlNode". Zahvaljujući toj konverziji možemo pozvati naredbu *GetNode* koja za trenutni element vraća njegov objekt tipa *XmlNode*.

Potom s dohvaćenim elementom možemo raditi što god želimo. U gornjem primjeru odlučili smo se na promjenu atributa "Ime". Njega postavljamo na redni broj trenutnog elementa među dohvaćenima – prvi će korisnik tako imati korisničko ime "1", drugi "2", itd.

XML serijalizacija

Kako je XML postao glavni format za razmjenu informacija, u .NET je ugrađena podrška za serijalizaciju i deserijalizaciju objekata u XML format. O čemu se zapravo radi? Serijalizacija vam daje mogućnost da bilo koji objekt spremite u XML obliku i, primjerice, spremite u neku datoteku na disku. Deserijalizacija vam pak omogućava da pročitate taj XML objekt i iz njega stvorite originalni objekt.

Za serijalizaciju i deserijalizaciju ključan je *System.Xml.Serialization namespace*. U njemu se nalaze metode za ispravno upravljanje različitim objektima i njihovu (de)serijalizaciju. Sve ćemo najbolje pokazati na primjeru objekta *DataSet* s kojim smo se družili u 9. poglavlju o ADO.NET-u.

Recimo da izrađujete aplikaciju koja omogućava uređivanje podataka iz baze. Sve obavljate pomoću *DataSet* objekta u kojem je spremljena kopija podataka. Po završetku uređivanja sve promjene spremate u bazu podataka. No što ako se desi nepredviđen slučaj – baza je nedostupna, trenutno se administrira ili je nedostupan poslužitelj u mreži na kojem se baza nalazi? Zar ćete dopustiti da sve promjene budu izgubljene ili ćete natjerati korisnika da pokušava spremiti podatke sve dok veza s bazom ne bude ispravna?

Rješenje ove situacije je jednostavno i nudi vam se u obliku XML serijalizacije. Po završetku rada aplikacija će pokušati spremiti podatke u bazu – u slučaju da ne uspije, jednostavno će serijalizirati objekt *DataSet* u kojem se nalazi kopija podataka. Spremit će ga negdje na disk i po sljedećem pokretanju aplikacije će ga učitati i ponovno pokušati spremiti u bazu. Ako i tada ne uspije, nema problema – aplikacija može i dalje raditi s lokalnom kopijom podataka spremljenom u serijaliziranom XML dokumentu sve dok se ne uspostavi veza s bazom.

Pri radu s narednim primjerima ne zaboravite uključiti *System.Xml.Serialization namespace*:

```
using System.Xml.Serialization;
```

U *DataSet* objekt u varijabli *dataSet11* učitali smo podatke dohvaćene preko objekta *DataAdapter* jednostavnom SQL naredbom izvršenom nad bazom Northwind, s kojom smo se družili i u 9. poglavlju.

```
SELECT EmployeeID, LastName, FirstName, BirthDate FROM Employees
```

Radi se o jednostavnom upitu koji dohvaća osnovne podatke o zaposlenicima. Dakle, u svojoj aplikaciji ste napunili *DataSet*. Evo kako biste taj *DataSet* serijalizirali u neku datoteku na disku:

```
XmlSerializer ser = new XmlSerializer(dataSet11.GetType());
XmlTextWriter xml = new XmlTextWriter("podaci.xml", System.Text.Encoding.UTF8);

ser.Serialize(xml, dataSet11);

xml.Close();
```

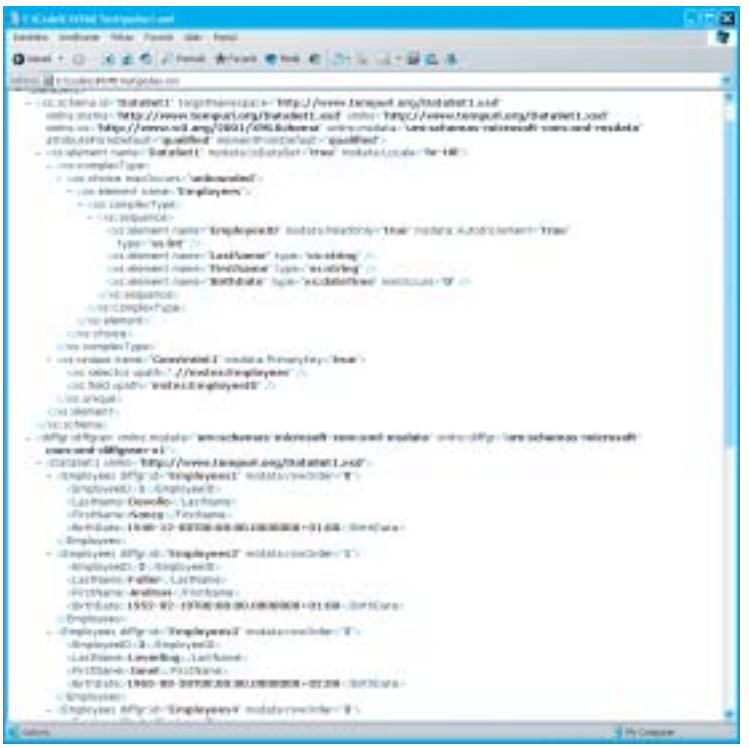
Prvo smo stvorili objekt tipa *XmlSerializer* i namjestili ga da radi s *DataSet* objektima. To smo učinili proslijedivši mu kao parametar u konstruktoru tip objekta koji će trebati serijalizirati, a njega smo pak dohvatili pozivom metode *GetType* nad objektom s kojim radimo.

Potom smo stvorili objekt za zapisivanje XML dokumenata tipa *XmlTextWriter* – kao prvi parametar zadali smo mu datoteku u koju će zapisati, a kao drugi *encoding*. Potom smo pozvali metodu *Serialize* objekta *XmlSerializer* – kao prvi parametar smo joj zadali *XmlTextWriter* koji određuje gdje će se zapisati podaci, a kao drugi sam objekt koji treba serijalizirati.

Na kraju zatvaramo *XmlTextWriter* i podaci se zapisuju u datoteku. Na slici 11-5 možete vidjeti strukturu generirane XML datoteke prikazane u Internet Exploreru, a posebnu pozornost obratite na XML shemu podataka koja se nalazi na početku dokumenta.

Naravno, sama serijalizacija vam ne bi previše vrijedila bez mogućnosti deserijalizacije. Ona je, u slučaju *DataSet* objekata, mnogo jednostavnija. Primjerice, imate li u prozorskoj aplikaciji *DataGrid* u kojem se prikazuju podaci i u njemu želite prikazati podatke iz *DataSeta*, evo kako biste to učinili:

```
dataSet11.ReadXml("podaci.xml");
dataGrid1.DataSource = dataSet11;
dataGrid1.DataMember = "Employees";
```



Slika 11-5: ***Serijalizirani DataSet u generiranoj XML datoteci***

Jednostavno u objekt *dataSet11* učitavamo sadržaj XML datoteke pozivom naredbe *ReadXml*. Dvije sljedeće naredbe su otprije poznate – povezujemo objekt *DataSet* s *DataGridom* i namještamo ga da prikazuje podatke iz tablice *Employees* odnosno *Employees* elemente iz učitanog XML dokumenta.

Serijalizacija vlastitih objekata

Osim što imate mogućnost serijalizacije svakog gotovog objekta u .NET-u, nudi vam se vrlo korisna mogućnost serijalizacije vlastitih objekata nastalih iz posebnih klasa. Pogledajte sljedeći primjer klase s različitim elementima na kojima će biti objašnjene mogućnosti serijalizacije:

```
[XmlRoot("MojaKlasaXML")]
public class MojaKlasa
{
    [XmlAttribute]
    public int Broj = 0;

    private string _Ime = "";
```

```
        public string Ime
        {
            get { return this._Ime; }
            set { this._Ime = value; }
        }

        private string _Prezime = "";
        [XmlIgnore]
        public string Prezime
        {
            get { return this._Prezime; }
            set { this._Prezime = value; }
        }

        public MojaKlasa()
        {
        }

        public MojaKlasa(int MojBroj)
        {
            Broj = MojBroj;
        }
    }
```

Dakle, radi se o klasi *MojaKlasa* koja ima cjelobrojnu varijablu *Broj*, znakovni niz *Ime* (realiziran preko metoda *get* i *set* i privatne varijable *_Ime*), znakovni niz *Prezime* (također realiziran preko metoda *get* i *set* i privatne varijable *_Prezime*). U klasi su definirana i dva konstruktora – jedan *defaultni* i jedan koji prima cjelobrojni parametar i upisuje ga u varijablu *Broj*.

Klasa koju serijalizirate obavezno mora imati definiran *defaultni* konstruktor, tj. onaj koji ne prima niti jedan parametar jer se on koristi u procesu serijalizacije odnosno deserijalizacije. Takav konstruktor je definiran i u gornjem primjeru – ne sadržava nikakav kôd i nema funkcionalnost, ali mora postojati.

Unutar klase možete vidjeti i niz direktiva koje određuju ponašanje elemenata pri serijalizaciji, a one su objašnjene u tablici 11-7.

Tablica 11-7:
Direktive koje određuju ponašanje elemenata pri serijalizaciji

Direktiva	Opis
XmlAttribute	član označen ovom direktivom bit će serijaliziran kao atribut u XML dokumentu
XmlElement	član označen ovom direktivom bit će serijaliziran kao element u XML dokument (*defaultno* ponašanje)
XmlIgnore	član označen ovom direktivom ignorirat će se pri serijalizaciji i neće biti uključen u generirani XML dokument
XmlRoot	opisuje korijenski element XML dokumenta (direktiva je primjenjiva isključivo na klasu)

Sad kad znate što koja direktiva iz tablice 11-7 znači, lako je razumjeti opis klase. Tako će objekt te klase biti serijaliziran u XML objekt koji ima korijenski element imena "MojaKlasaXML". Varijabla *Broj* će, zbog direktive *XmlAttribute*, biti stavljena kao atribut u glavni element. Isto tako, varijabla *Prezime* će se ignorirati jer je ispred nje postavljena direktiva *XmlIgnore*.

Serijalizacija objekta neke klase ne uključuje članove koji nisu označeni s *public* te metode. Tako se ne morate brinuti da će se, primjerice, serijalizirati privatna varijabla *_Ime*, a metode se ionako ne mogu serijalizirati jer ne sadržavaju nikakve podatke.

Objekt klase *MojaKlasa* serijalizirate na sličan način kao i *DataSet* objekte – pogledajte primjer:

```
MojaKlasa obj = new MojaKlasa(15);
obj.Ime = "Marko";
obj.Prezime = "Marić";

XmlSerializer ser = new XmlSerializer(obj.GetType());
XmlTextWriter xml = new XmlTextWriter("podaci.xml", System.Text.Encoding.UTF8);

ser.Serialize(xml, obj);
xml.Close();
```

Na početku stvaramo objekt i preko konstruktora mu vrijednost varijable *Broj* postavljamo na 15. Također, postavljamo mu vrijednosti za varijable *Ime* i *Prezime*. Ostatak kôda je isti kao i kod serijalizacije *DataSeta*: stvara se novi *XmlSerializer* objekt, zatim *XmlTextWriter* koji određuje gdje će se spremiti serijalizirani objekt, te se poziva metoda *Serialize*.

Pogledamo li nakon izvršavanja prethodnog kôda u datoteku "podaci.xml", ona će imati sljedeći sadržaj:

```
<?xml version="1.0" encoding="utf-8" ?>
<MojaKlasaXML xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" Broj="15">
   <Ime>Marko</Ime>
</MojaKlasaXML>
```

Kao što smo i očekivali, objekt se serijalizirao u XML dokument s korijenskim elementom imena "MojaKlasaXML", a kao njegov atribut je postavljena i vrijednost varijable *Broj*. Unutar tijela XML dokumenta nalazi se element *Ime,* koji sadržava vrijednost varijable *Ime*, no nema vrijednosti varijable *Prezime*, jer je kod njene definicije u klasi postavljena direktiva *XmlIgnore*.

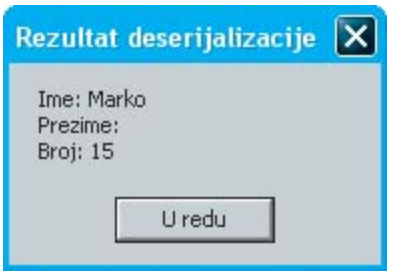
Deserijalizacija ovakvog objekta ide na malo drugačiji način nego kôd objekta *DataSet*, jer vaš objekt nema metodu *ReadXml*. No, na svu sreću, on uopće nije kompliciran – želite li u neki objekt klase *MojaKlasa* učitati podatke iz XML datoteke, evo kako biste to učinili.

```
MojaKlasa novi = new MojaKlasa();
XmlSerializer deser = new XmlSerializer(novi.GetType());
XmlTextReader x = new XmlTextReader("podaci.xml");
novi = (MojaKlasa) deser.Deserialize(x);

MessageBox.Show("Ime: " + novi.Ime + "\nPrezime: " + novi.Prezime + "\nBroj: " +
novi.Broj.ToString(), "Rezultat deserijalizacije");
```

Prvo stvaramo novi objekt klase *MojaKlasa* i pritom koristimo *defaultni* konstruktor, tj. onaj bez parametara, jer želimo da novi objekt bude *prazan*, bez postavljenih vrijednosti. Zatim inicijaliziramo objekt tipa *XmlSerializer*, baš kao u procesu serijalizacije.

No za razliku od serijalizacije kad smo koristili *XmlTextWriter*, jer smo zapisivali XML dokument, u procesu deserijalizacije koristimo *XmlTextReader*, jer čitamo sadržaj nekog XML dokumenta. Potom pozivamo metodu *Deserialize* koja kao parametar prima objekt iz kojeg čitamo XML dokument i dohvaćeni deserijalizirani objekt pretvaramo u objekt tipa *MojaKlasa* jednostavnim *castanjem* u taj tip.

No znakovito je ispisivanje informacija o deserijaliziranom objektu – kao što je vidljivo na slici 11-6, nije dohvaćena vrijednost za varijablu *Prezime*, što je logično, jer ona nije ni bila serijalizirana.



Slika 11-6:
Ispis informacija o deserijaliziranom objektu

Kao što ste mogli vidjeti u ovom poglavlju, XML vam uvelike može olakšati izradu aplikacija. Zahvaljujući mogućnostima serijalizacije objekata u XML, imate otvorene mogućnosti spremanja objekata na lokalni disk za kasniju uporabu. U narednom poglavlju čekaju vas web-servisi koji se u potpunosti temelje na XML-u i SOAP-u, posebnoj gramatici XML-a.

12. POGLAVLJE

Web-servisi

U ovom poglavlju:

- Standardi vezani uz web-servise
- Stvaranje web-servisa
- Opcije web-metoda
- Isprobavanje web-servisa u pregledniku
- Korištenje web-servisa u .NET aplikacijama
- Ručno i automatsko generiranje proxy-klasa

Dosad ste naučili izrađivati *desktop* aplikacije i web-aplikacije, a sad slijedi prava poslastica – web-servisi. Krenimo odmah s objašnjenjem: na web-servise se može gledati kao na svojevrsne potprograme smještene na Webu. Dakle, radi se o programima smještenima na nekom web-poslužitelju (baš kao i obične web-aplikacije) koje možete koristiti putem standardnog protokola HTTP.

Njima tako možete pristupati putem Weba, koristiti ih, zadavati im upite, a oni će vam na isti način vraćati odgovore. No web-servisi nisu izmišljeni da bi s njima direktno komunicirao korisnik, već neki drugi program. Kao što je rečeno, na njih se može gledati kao na potprograme koje će druge aplikacije pozivati i koristiti u svom radu. Zahvaljujući činjenici

da se komunikacija s web-servisima, kao što im i ime kaže, odvija preko Weba, njih mogu koristiti baš svi.

Dajmo odmah jedan praktičan primjer: banka objavi svoj web-servis koji ima jednostavan zadatak – vraćati tečajnu listu za određeni dan. Web-servis može primati jedan parametar, i to datum za koji se traži tečajna lista. No sad tu tečajnu listu može koristiti bilo koja aplikacija s pristupom na Internet – jednostavno se poveže s web-servisom, zada mu upit i dobije odgovor s tečajnom listom koju tad može prikazati u svojoj aplikaciji.

Aplikacije Microsoft Officea 2003 u sebi također imaju ugrađenu podršku za vanjske web--servise, pa tako direktno iz Worda ili Excela možete pristupati dostupnim web-servisima. U Hrvatskoj je već načinjeno nekoliko takvih servisa, pa tako možete pretraživati tečajne liste, vozni red vlakova, pozivne brojeve gradova i država itd.

U ovom poglavlju pokazat ćemo vam koliko je lako izrađivati web-servise i koliko je lako koristiti ih i pozivati iz vlastitih aplikacija. Dakle, pokazat ćemo dvije strane priče – prvo ćemo se postaviti u kožu autora i programera web-servisa, a zatim u situaciju programera aplikacije koja želi iskoristiti neki web-servis.

Web-servisi ne trebaju biti javni odnosno ne trebate ih dati svima na korištenje. Najčešće ćete izraditi vlastite web-servise koje ćete koristiti samo iz vlastite aplikacije, bila ona web-aplikacija, prozorska aplikacija ili pak aplikacija za mobilne uređaje. Pritom će glavna uloga web-servisa biti da objedinjavaju neku programsku logiku odnosno predstavljaju već spomenuti *potprogram* koji možete pozivati iz bilo kojeg dijela svoje aplikacije.

Standardi

No dobro, u čemu je tajna web-servisa? Vrlo jednostavno – tajna je u njihovoj standardiziranosti. Naime, web-servisi nisu uopće ovisni o platformi. Recimo to najjednostavnijim rječnikom: web-servis možete izgraditi na IBM-ovoj platformi i zatim ga koristiti iz .NET aplikacija; web-servis možete izgraditi na .NET-u, a zatim ga koristiti iz Oracle aplikacija; web-servis možete izgraditi uz pomoć Oracleove tehnologije, a zatim ga koristiti iz aplikacija izrađenih u IBM-ovim alatima.

Recimo to ispravno – web-servisi nisu nečija tehnologija, već dogovoreni standard koji omogućava svima da ih koriste i time omogućavaju izradu pravih distribuiranih sustava. Pogledamo li malo šire, veoma rijetko ćemo pronaći tehnologije koje se koriste na svim platformama, što čini web--servise doista uspješnima.

Nigdar ni bilo...

Nekoć davno, u doba zmajeva i prinčeva, nije bilo web-servisa, no oduvijek postoji ideja i želja za takozvanim distribuiranim aplikacijama. Kao što im i ime kaže, radi se o aplikacijama čija je funkcionalnost *podijeljena* na više lokacija i više programa. Jedan od mnogih koraka u tom smjeru bio je i Distributed COM (DCOM), čiji je zadatak bio da funkcionalnost programa bude distribuirana kroz mrežu, no opet dostupna svim zainteresiranim programima. DCOM je bio izgrađen na temeljima *remote procedure call* (RPC) arhitekture, što je za posljedicu imalo i niz nedostataka.

Kao prvo, DCOM i RPC su bili mnogo prikladniji za intranetska okruženja, a ne za Internet. *Portovi* kojima komuniciraju DCOM i RPC najčešće nisu otvoreni na korporacijskim mrežama, već su zaštićeni vatrozidovima. Iako to nije bio problem sve do popularizacije Interneta, danas je to ipak preveliko okruženje i svaka distribuirana tehnologija mora bez problema raditi preko Interneta (budimo potpuno precizni – za DCOM je postojao COM Internet Services koji je za komunikaciju koristio HTTP *port* 80 i tako izbjegavao vatrozide, no to ipak nije bilo rješenje svih problema).

Drugi njihov nedostatak bio je utjecaj na same aplikacije. Njihovo korištenje bilo je jako teško i komplicirano te je najčešće zahtijevalo promjenu arhitekture aplikacija. Kako je DCOM nadogradnja COM (Component Object Model) tehnologije, koja je, kao što smo u prvim poglavljima objasnili, zastarjela pojavom .NET-a zbog svojih nedostataka, i DCOM je patio od tih istih nedostataka – ne baš jednostavnog korištenja, problemima s registriranjem ispravnih verzija komponenti i slično.

Na samom kraju, najveći nedostatak iz današnje perspektive bilo je to da DCOM nije bio neovisan o platformi, već je podrazumijevao Microsoftove Windowse. Web-servisi, naravno, adresiraju svaki od navedenih nedostataka i nameću se kao rješenje budućnosti (a i današnjice) za distribuirane sustave.

Kao što smo rekli – web-servise možete izrađivati na nizu različitih platformi, a cilj ovog poglavlja je pokazati vam koliko je to jednostavno na Microsoft .NET platformi i u razvojnom okruženju Visual Studija .NET.

Razlog tolike standardnosti web-servisa i njihove opće prihvaćenosti leži u dvjema tehnologijama na kojima se oni temelje. Jedna je već spomenuta – HTTP protokol koji je temelj Weba, a druga je obrađena u prethodnom poglavlju – XML.

XML i HTTP spojeni zajedno rezultirali su novim standardom, Simple Object Access Protocol (SOAP). Za njega je zadužen W3C (World Wide Web Consortium), neovisno tijelo zaduženo za razvoj Interneta i donošenje novih standarda. SOAP koristi XML za slanje poruka preko HTTP-a, a zapravo se radi o posebnoj XML gramatici, kao što ćete vidjeti kasnije.

Uz SOAP ruku pod ruku ide i još nekoliko standarda – najvažniji je svakako WSDL ili Web Services Description Language. Njega su razvili Microsoft, IBM i drugi proizvođači ujedinjeni oko inicijative web-servisa, a radi se prvenstveno o XML-shemi koja opisuje neki web-servis, sve njegove metode i njihove parametre. Zahvaljujući toj shemi moguće je komunicirati s web-servisom, jer je poznato koju funkcionalnost sadržava te kako se ta funkcionalnost može iskoristiti i pozvati iz druge aplikacije.

Tu je i UDDI – Universal Description, Discovery and Integration, otvoreni *framework* za opisivanje web-servisa i njihovu međusobnu integraciju i povezivanje. UDDI je također rezultat IBM-ova i Microsoftova prijedloga, a više informacija o njemu možete naći na http://www.uddi.org/.

Slika 12-1: ***Web-site www.uddi.org sadržava detaljne informacije o UDDI-u.***

Uz sve opisane standarde, pri izradi web-servisa u .NET-u susrest ćete se i .disco datotekama odnosno *Discoveryem*, Microsoftovim protokolom za XML dokumente koji sadrže linkove na druge resurse koji opisuju web-servis.

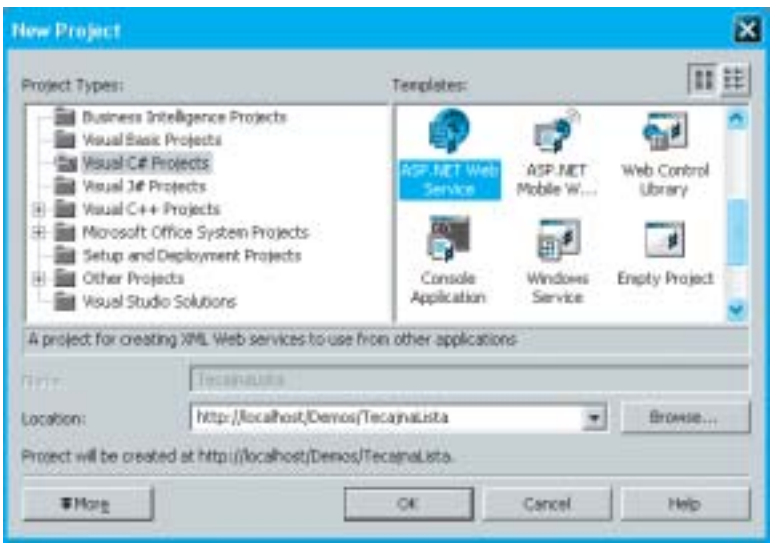
Iako SOAP postoji još od 1999. godine, dosad nije bilo puno programera koji su ga znali iskoristiti za stvaranje SOAP aplikacija. Dolaskom .NET-a i integrirane podrške za izradu web-servisa u Visual Studiju, za razvoj web-servisa koji, kao što je rečeno, koriste SOAP protokol, gotovo da i nećete morati poznavati tajne SOAP-a. To je, naravno, djelomično istina, jer ćete vi, kao savjestan programer željan novih znanja i iskustava, rado proučavati SOAP poruke kako se budete bavili web-servisima i znanje o SOAP-u ćete nesvjesno usvojiti u svakodnevnom radu.

Eto, sad kad ste uvjereni da su web-servisi *way to go*, ne samo zbog svoje opće prihvaćenosti i nevezanosti isključivo uz Microsoft, vrijeme je da se bacimo na pravi posao – izradu web-servisa. Bez straha, izrada web-servisa u Visual Studiju .NET je doista lagana, a ako ste svladali prethodna poglavlja i udomaćili se u programiranju .NET aplikacija, za vas će ostatak ovog poglavlja biti mačji kašalj.

Izrada web-servisa

Nakon svega što ste vidjeli u ovoj knjizi, vjerojatno vas neće posebno iznenaditi spomenuta lakoća njihove izrade u Visual Studiju .NET, jer ćete takvo što ipak očekivati. Dakle, njihova izrada se ni po čemu ne razlikuje od već opisane izrade prozorskih ili web-aplikacija.

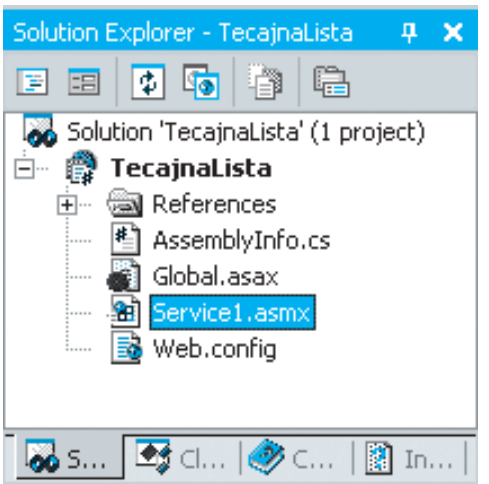
Stoga se odmah upustimo u izradu web-servisa: u Visual Studiju odaberite *File – New – Project* te zatim pod *Visual C# Projects* pronađite *ASP.NET Web Services*. To je pravi trenutak da objasnimo zašto se tako zovu odnosno zašto riječ ASP.NET u njihovu nazivu. Naime, web-servis će biti smješten u nekom virtualnom direktoriju na Internet Information Services (IIS) poslužitelju, što je identično standardnim ASP.NET web-aplikacijama jer će se njima pristupati preko Weba.



Slika 12-2: ***Stvaranje novog web-servisa***

Stoga u polje *Location* upisujete lokaciju na poslužitelju gdje će web-servis biti smješten, a to ujedno i predstavlja njegovo ime. U nastavku ćemo se baviti izradom jednostavnog servisa, pojednostavljene tečajne liste, stoga smo ga nazvali "TecajnaLista".

Nakon što kliknete OK, Visual Studio .NET će za vas automatski stvoriti virtualni direktorij i namjestiti sve potrebne datoteke.



Slika 12-3: ***Početne datoteke novog web-servisa***

Primijetite da među datotekama u direktoriju web-servisa postoji i web.config datoteka. Nju ste već prije upoznali u poglavlju o ASP.NET-u, a samo ukratko podsjetimo – služi za osnovnu konfiguraciju postavki web-aplikacije, a njeno postojanje uz web-servis dodatno objašnjava zašto ih u Visual Studiju .NET zovu "ASP.NET Web Services". Naravno, kao i kod ASP.NET aplikacija, web.config datoteka ima istu namjenu, te je stoga nećemo dodatno objašnjavati.

Web-servisi su u .NET-u predstavljeni kao datoteke s ekstenzijom .asmx. Pogledajte među datotekama na slici 12-3 i vidjet ćete jednu imena *Service1.asmx*. Svaka datoteka s .asmx ekstenzijom je jedan web-servis, a on može sadržavati više funkcionalnosti. ASMX datoteke, baš kao i ASPX za web-stranice ili ASCX za web-kontrole, kontrolira IIS i izvršava njihov kôd.

Za početak, preimenujmo u *Solution Exploreru* datoteku *Service1.asmx* – označite je, pritisnite F2 i upišite novo ime, primjerice "Valute.asmx" pa je, ako nije već otvorena, dvaput kliknite i otvorite.

Za web-servise je ključno razumjeti da oni nemaju sučelje i da s njima ne radite kao s običnim ASP.NET stranicama. Tako će i datoteka *Valute.asmx* sadržavati sljedeći kôd:

```
<%@ WebService Language="c#" Codebehind="Valute.asmx.cs"
Class="TecajnaLista.Service1" %>
```

Nju ćete standardno otvoriti u grafičkom načinu rada i tamo će vas dočekati poruka koja kaže da, ako želite dodati komponente web-servisu, trebat ćete ih povući iz *Server Explorera* ili *Toolboxa* i iskoristiti *Properties* prozor za njihovo namještanje, no isto tako, ako želite stvoriti nove metode, trebat ćete se prebaciti u prikaz kôda.

Prevedeno, kako web-servisi nemaju svoje sučelje kojem biste dodali kontrole poput tekstualnih polja za unos ili padajućih izbornika, jedino što možete raditi jest pisati kôd. No isto tako, možete iz *Toolboxa* dodavati komponente za spajanje na baze podataka na isti način, što je i objašnjeno u 9. poglavlju.

Što se tiče sadržaja ASMX datoteke, kao što je i vidljivo u prethodnom ispisu, tu se nalazi samo uputa o web-servisu, a kompletan kôd će biti sadržan u datoteci s pozadinskim kôdom s ekstenzijom .asmx.cs. Atribut *Class* određuje ime klase web-servisa. Pri prvom pokretanju i pristupanju web--servisu, mehanizam ASP.NET-a će pretražiti direktorij *bin* u potrazi za *assemblyjem* koji sadrži navedenu klasu.

Pisanje kôda

Kao što smo rekli, odlučili smo se za izradu jednostavne tečajne liste. U pravom svijetu vi ćete te podatke dohvaćati iz baze ili neke XML datoteke, no zbog jednostavnosti i da bi vam fokus ostao na najzanimljivijoj stvari – samoj izradi web-servisa, odlučili smo se za lakši pristup.

Ako baš želite izrađivati web-servis tečajne liste, Hrvatska narodna banka svaki dan objavljuje tečajnu listu u posebnom formatiranom zapisu prilagođenom za automatsku obradu. Za više informacija posjetite http://www.hnb.hr/tecajn/htecajn.htm. Te datoteke možete automatski preuzimati i iz njih programski izvlačiti odgovarajuće podatke.

Prebacite se u prikaz kôda web-servisa – kliknite na link "click here to switch to code view" u prikazu ASMX datoteke ili desnim gumbom na Valute.asmx i odaberite *View Code*. U kodu ćete tako pronaći komentiranu sljedeću metodu:

```
[WebMethod]
public string HelloWorld()
```

```
{
    return "Hello World";
}
```

Njena struktura sasvim dobro objašnjava dijelove neke metode web-servisa. Da biste nekoj funkcionalnosti, tj. metodi web-servisa mogli pristupati putem Weba, ona mora biti označena s [WebMethod], a prava pristupa joj moraju biti postavljena na *public*. Takva se metoda može pozivati kao metoda web-servisa, a u gornjem primjeru ona samo vraća poruku "Hello World".

Naravno, kao i kod programiranja drugih tipova aplikacija, unutar web-servisa možete imati velik broj metoda koje se interno koriste, no nisu označene s [WebMethod] pa im nije moguće pristupati izvana preko Weba. U njih možete staviti dijelove kôda koji se često koriste iz različitih metoda web-servisa.

No mi ćemo ipak napisati svoju verziju jednostavne web-metode. Napisat ćemo metodu koja će iznos u nekoj drugoj valuti pretvarati u kune. Primat će dva parametra – jedan će označavati iznos, a drugi valutu. Primjerice, pozovemo li metodu s parametrima (100, "EUR"), rezultat će biti 750.11 (naravno, kuna).

Evo kako bi i izgledao kôd takve metode (još jednom napomena – zbog jednostavnosti smo *hard-kodirali* odnose tečajeva, no u stvarnoj situaciji vi biste te podatke vukli iz baze podataka ili nekog drugog izvora, poput prije spomenute datoteke koju objavljuje HNB).

```
[WebMethod]
public double PretvoriValutu(double Kolicina, string Valuta)
{
    double faktor, Kuna;
    switch (Valuta)
    {
        case "EUR": faktor = 7.501118; break;
        case "USD": faktor = 6.115374; break;
        case "CHF": faktor = 4.787540; break;
        case "GBP": faktor = 11.094687; break;
        default: faktor = 0; break;
    }
    Kuna = Math.Round(faktor * Kolicina, 2);
    return Kuna;
}
```

Kôd je uistinu jednostavan – metoda prima dva parametra te u ovisnosti o drugom parametru izračunava faktor. Potom množi proslijeđeni iznos (varijabla *Kolicina*) s faktorom i zaokružuje rezultat na dvije decimale. Taj se rezultat (varijabla *Kuna*) vraća kao rezultat metode.

Imenski prostor web-servisa

Pri izradi web-servisa preporučuje se dodijeliti mu odgovarajući *namespace*. To može biti vaša internetska adresa na kojoj će biti smješten web-servis, ime tvrtke ili bilo što drugo što bi ga moglo identificirati. *Namespaceovi* web-servisa tako često podsjećaju na URL-ove, no oni ne moraju ukazivati na stvarne resurse na Internetu.

Ukoliko svom web-servisu ne dodijelite *namespace*, dobit će *defaultni*, "http://tempuri.org/". On je prikladan za web-servise koji se još razvijaju i nisu objavljene njihove finalne verzije, no čim završite s izradom web-servisa, preporučuje se namještanje *namespacea*. Taj *namespace* će biti korišten u svim SOAP porukama i služi za lakše identificiranje vašeg servisa.

***Namespace* se postavlja klasi web-servisa. Prebacite li se u kôd, na početku ćete vidjeti naziv klase (imat će ime "Service1", ako ste sve radili kao što je opisano u knjizi), a iznad te linije trebate dodati *namespace*, primjerice:**

```
[WebService(Namespace="http://NE
T/Poglavlje12/TecajnaLista")]
public class Service1 :
System.Web.Services.WebService
{
    // itd.
```

***Namespace* se dodaje unutar *WebService* direktive. U našem smo primjeru dodali *namespace* koji nam pomaže u snalaženju – podsjeća na URL, no on to nije. Naravno, vi *namespaceu* možete dati bilo koji oblik i bilo koju vrijednost, to je potpuno na vama.**

Klasa *Service1* koja u sebi sadržava svu funkcionalnost web-servisa, nasljeđuje *System.Web.Services.WebService*. U klasi *WebService* se tako nalaze članovi potrebni za ispravan rad web-servisa. U njoj je i podrška za rad sa *sessionima* te objekti *Server* i *User*, baš kao i u web-aplikacijama.

Opcije web-metoda

Kao što ste vidjeli, da bismo neku metodu učinili dijelom javne i svima dostupne funkcionalnosti web-servisa, označili smo je s [WebMethod]. Sama direktiva ima šest opcija pomoću kojih se dodatno može upravljati načinom rada servisa.

Pri čitanju dokumentacije vezane uz web-servise, susrest ćete se s nekoliko sličnih akronima: URI, URL i URN. URI ili Uniform Resource Identifier je niz koji jedinstveno identificira neki resurs na mreži. Postoje dva tipa URI-ja: Uniform Resource Locator (URL) i Uniform Resource Name (URN). URL je zadan prefiksom koji određuje protokol, adresom servera ili IP adresom, portom i putem do resursa (radi se o standardnoj adresi, primjerice http://www.server.com-/aplikacija/). URN je pak običan opisni niz koji podsjeća na adresu, primjerice NET12://tecaj-na_lista. Pri obradi URN-a, aplikacija može znati da NET12 odgovara 12. poglavlju knjige o .NET-u te da se u njemu nalazi opis tečajne liste. Dakle, radi se o opisu – URN ne ukazuje na stvarni resurs na Internetu.

BufferResponse. Njena *defaultna* postavka je *true* jer je omogućeno *bufferiranje* odgovora web-servisa odnosno njegovo slanje *u komadu*. Ukoliko ga pak postavite na *false*, odgovor web-servisa će se slati u dijelovima velikim 16 KB. Naravno, preporučljivo je ostaviti *defaultnu* opciju i ne mijenjati ništa, jer se slanjem u jednom dijelu smanjuje opterećenost servera i ubrzava prijenos podataka.

```
[WebMethod(BufferResponse=false)]
```

CacheDuration. Odgovori odnosno rezultati poziva web-servisa mogu se *cacheirati*, baš kao i rezultati ASP.NET skripti. Vrijednost ovog atributa određuje vrijeme *cacheiranja* odgovora u sekundama. *Defaultno* vrijeme je 0, odnosno ništa se ne *cacheira*. No ukoliko imate određen set parametara koji mogu biti proslijeđeni web-servisu i koji uvijek rezultiraju istim rezultatima, isplati se uključiti *cacheiranje*.

Evo i primjera kako namjestiti da se rezultati metode *cacheiraju* i čuvaju 5 minuta. Ukoliko se za to vrijeme dogodi poziv s istim parametrima, on neće izvršiti metodu web-servisa, već će odmah dobiti spremljeni odgovor.

```
[WebMethod(CacheDuration=300)>
```

Description. Unutar ovog parametra navodite opis web-servisa koji se pojavljuje na stranici za pomoć. Kako se radi o tekstualnom parametru, njegovu vrijednost navodite unutar navodnika.

```
[WebMethod(Description="Tečajna lista vraća podatke…")]
```

EnableSession. Omogućava korištenje *sessiona* te može imati vrijednost *true* ili *false*. Ukoliko omogućite *sessione*, možete im pristupati preko *HttpContext.Current.Session* ili preko *WebService.Session*. Način korištenja *sessiona* isti je kao i u web-aplikacijama.

```
[WebMethod(EnableSession=true)]
```

MessageName. Ukoliko koristite preopterećene metode (prisjetimo se 6. poglavlja – preopterećene metode su one metode koje imaju isto ime, no drugačiji potpis, tj. različite parametre i tip), korištenjem *MessageName* možete namjestiti drugačije ime za svaku metodu. Po *defaultu* je vrijednost parametra *MessageName* jednaka imenu metode, no ukoliko imate više istoimenih metoda, možete im definirati *aliase*.

```
[WebMethod(MessageName="TecajKune")]
```

TransactionOption. Ovaj parametar omogućava sudjelovanje metode web-servisa kao korijenski objekt u transakciji. Za postavljanje njegove vrijednosti koristi se pobrojan niz *TransacionOption*, a možete iskoristiti dvije mogućnosti – ili web-servis ne može sudjelovati u transakciji (*Disabled*, *NotSupported* ili *Supported*) ili se stvara nova transakcija (*Required*, *RequiresNew*). *Defaultna* vrijednost je *TransactionOption.Disabled*. Da biste mogli koristiti transakcije, u web-servisu trebate uključiti referencu na *System.EnterpriseServices.dll* odnosno uključiti *System.Enterprise namespace*..

```
[WebMethod(TransactionOption=TransactionOption.RequiresNew)]
```

Kako je HTTP protokol *stateless* odnosno prethodni koraci i operacije se ne pamte, web-servisi ne mogu punopravno sudjelovati u postojećim transakcijama. Primjerice, u slučaju *rollbacka* odnosno otkazivanja transakcije provjeravanje što je sve transakcija promijenila i otkazivanje svih operacija bi bilo problematično. No ipak, web-metode mogu pokretati nove transakcije – tako možete pratiti rad i izvršene naredbe u web-metodi i na njenom kraju (ili ranije), ako želite, otkazati sve promjene.

Kad se unutar web-metode pojavi iznimka koja se ne obradi ispravno, transakcija se odmah prekida. Ukoliko pak nema nikakve iznimke, transakcija se na kraju metode automatski izvršava. Izvršavanje transakcije znači da se sve akcije obavljene unutar metode apliciraju i primjenjuju odnosno izvršavaju. Ukoliko se pak pozove *SetAbort* metoda, transakcija se prekida i odustaje od izvršavanja metode – niti jedna linija kôda metode se neće doista izvršiti (ako se već neka naredba izvršila, ona će se otkazati i sve će se vratiti na prethodno stanje). Naravno, da biste mogli koristiti *SetAbort* metodu (i *SetComplete*), možete im pristupiti preko *ContextUtil* klase koja je definirana u *System.Enterprise namespaceu*.

Iako većinu ovih parametara nećete mijenjati, neke od njih ipak možete iskoristiti za metode svojih web-servisa. Tako ćete često iskoristiti *cacheiranje* podataka te davanje opisa samim meto-

dama. Evo i kako biste spojili dva gornja parametra i definirali da se odgovor metode web-servisa cacheira narednih 5 minuta te namjestili njen opis.

```
[WebMethod(CacheDuration=300, Description="Metoda pretvara iznos u stranoj valuti u kunsku protuvrijednost")]
```

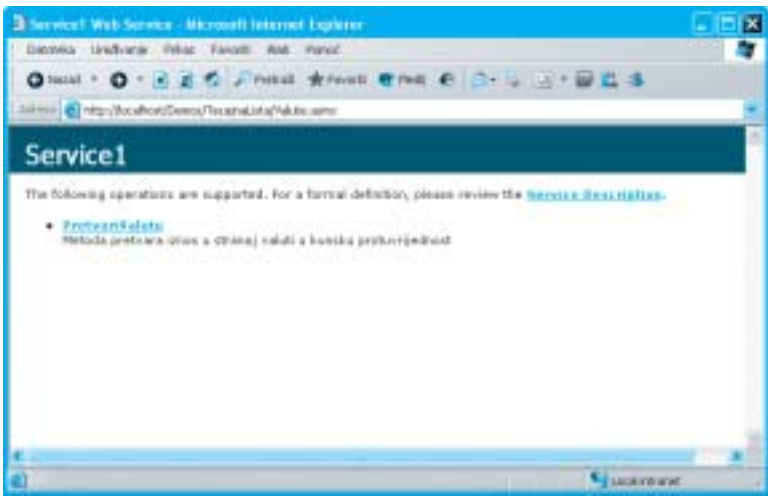
Atributi *WebMethod* direktive odvajaju se zarezima pa ih tako, ako želite, možete sve navesti.

Isprobavanje web-servisa

Nakon mukotrpne izrade web-servisa i dugog vremena prilagođavanja na njihov poseban način pisanja kôda (naravno, uočavate ironiju), vrijeme je za njihovo isprobavanje. Baš kao i pri izradi drugih tipova aplikacija, iskoristite opciju *Build* (izbornik *Build* – *Build Solution* ili pritisnite CTRL+SHIFT+B).

Kako ste na početku stvaranja web-servisa odabrali lokaciju na kojoj će se on nalaziti, na lokalnom serveru *localhost* u nekom virtualnom direktoriju, možete otvoriti preglednik i ručno upisati adresu. Ukoliko ste radili sve kao i u primjeru, web-servis će se nalaziti na adresi http://localhost/Demos/TecajnaLista/Valute.asmx. Primijetite da za adresu web-servisa upisujete točnu adresu ASMX skripte jer se u njoj nalazi web-servis.

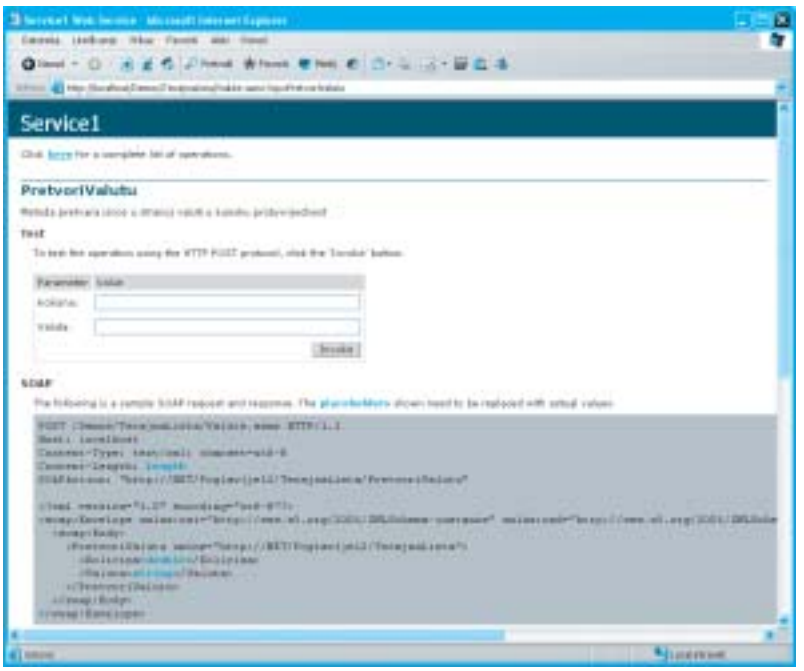
Naravno, ne trebate sami upisivati adresu. U Visual Studiju u izborniku *Debug* odaberite opciju *Start Without Debugging* (CTRL+F5) ili *Start* (F5) i otvorit će se Internet Explorer te automatski učitati web-servis.



Slika 12-4: ***Prikaz web-servisa u pregledniku***

Nakon pokretanja i učitavanja web-servisa, dočekat će vas sučelje poput onog na slici 12-4. Radi se o automatski generiranom sučelju koje prikazuje popis svih metoda web-servisa. Kako je u našem primjeru postojala samo jedna metoda označena s *WebMethod*, tako je samo ona ponuđena u sučelju (primijetite ispod nje opis koji smo naveli kao *Description* parametar u *WebMethod* direktivi), no da smo ih imali više, sve bi bile prikazane.

Klikom na tu metodu otvorit će vam se sučelje za komunikaciju s njom, kao na slici 12-5. Tu će se pojaviti tekstualna polja za unos svih parametara. Imena parametara su automatski izvučena iz kôda metode i naziva parametara koji joj se prosljeđuju.



Slika 12-5: ***Prikaz sučelja za isprobavanje metode web-servisa***

Uočite i adresu te metode koja neodoljivo podsjeća na ASP.NET skripte – kao parametar *op* proslijeđeno je ime metode koju želimo pozivati.

```
http://localhost/Demos/TecajnaLista/Valute.asmx?op=PretvoriValutu
```

Upišite u tekstualna polja parametre za isprobavanje metode. Primjerice, možete za parametar *Kolicina* upisati "500", a za parametar *Valuta* "USD". Pritisnite gumb *Invoke* i otvorit će se novi prozor s rezultatima poziva metode prikazan na slici 12-6.

Na stranici neke metode web-servisa prikazane su i SOAP-poruke koje služe za komunikaciju. Iako nije obavezno, bacite pogled na njih i proučite njihov sadržaj. SOAP je ipak preveliko područje da bismo ga obradili u ovoj knjizi, no o njemu postoji čitav niz zasebnih knjiga, koje vam preporučujemo. Zasad samo informativno pogledajte kako su te poruke građene i kako se obavlja komunikacija s web-servisom.

Složenac

Iako smo u našem primjeru pokazivali vraćanje jednostavnih tipova podataka zahvaljujući XML serijalizaciji objekata i SOAP-u koji može prezentirati kompleksne tipove podataka zbog internog korištenja XML shema, web-servis može vraćati objekte poput *DataSetova*, klasa i struktura. Primjerice, mogli smo napraviti web-servis koji bi vraćao *DataSet* – evo njegove skraćene verzije:

```
[WebMethod]
public DataSet
PretvoriValutu(double Kolicina,
string Valuta)
{
    DataSet ds = new DataSet();
    // punjenje DataSeta iz baze
    // podataka ili u samom kodu
    return ds;
}
```

Dakle, vi biste iz aplikacije koja koristi web-servis mogli pozvati gornju metodu i automatski biste u svojoj aplikaciji dobili rezultat u obliku objekta *DataSet*. Interno bi se konverzija odvijala dvaput – prvi put bi se *DataSet* iz web-metode pretvorio u XML u SOAP-poruci koja bi se poslala aplikaciji, zatim bi se u aplikaciji dobivena SOAP-poruka natrag pretvorila u objekt *DataSet* koji biste mogli normalno koristiti u svojoj aplikaciji. Vi pritom ne trebate ništa znati o tim konverzijama i na sve možete gledati s više razine – web-metoda vraća *DataSet* koji vi primate i normalno koristite u svojoj aplikaciji (u nastavku poglavlja bit će prikazano kako iz aplikacije pozivati web-servise i raditi s njihovim rezultatima).

Evo i kako izgleda objekt *DataSet* pretvoren u XML unutar SOAP-poruke – radi se o *DataSetu* s jednim objektom *DataTable* koji ima dva stupca (Valuta i Faktor) te dva zapisa s odgovarajućim vrijednostima. Uočite da je na njegovom početku prvo definirana XML shema podataka, a tek poslije stvarni podaci.

```
<?xml version="1.0"
encoding="utf-8" ?>
<DataSet
xmlns="http://NET/Poglavlje12/Te
cajnaLista">
  <xs:schema id="NewDataSet"
xmlns=""
xmlns:xs="http://www.w3.org/2001
/XMLSchema"
xmlns:msdata="urn:schemas-
microsoft-com:xml-msdata">
    <xs:element
name="NewDataSet"
msdata:IsDataSet="true"
msdata:Locale="hr-HR">
```

```
            <xs:complexType>
              <xs:choice maxOccurs="unbounded">
                <xs:element name="Tečajna">
                  <xs:complexType>
                    <xs:sequence>
                      <xs:element name="Valuta" type="xs:string"
                          minOccurs="0" />
                      <xs:element name="Faktor" type="xs:double"
                          minOccurs="0" />
                    </xs:sequence>
                  </xs:complexType>
                </xs:element>
              </xs:choice>
            </xs:complexType>
          </xs:element>
        </xs:schema>
        <diffgr:diffgram xmlns:msdata="urn:schemas-microsoft-com:xml-msdata"
            xmlns:diffgr="urn:schemas-microsoft-com:xml-diffgram-v1">
          <NewDataSet xmlns="">
            <Tečajna diffgr:id="Tečajna1" msdata:rowOrder="0"
                diffgr:hasChanges="inserted">
              <Valuta>EUR</Valuta>
              <Faktor>7.501118</Faktor>
            </Tečajna>
            <Tečajna diffgr:id="Tečajna2" msdata:rowOrder="1"
                diffgr:hasChanges="inserted">
              <Valuta>USD</Valuta>
              <Faktor>6.115374</Faktor>
            </Tečajna>
          </NewDataSet>
        </diffgr:diffgram>
      </DataSet>
```

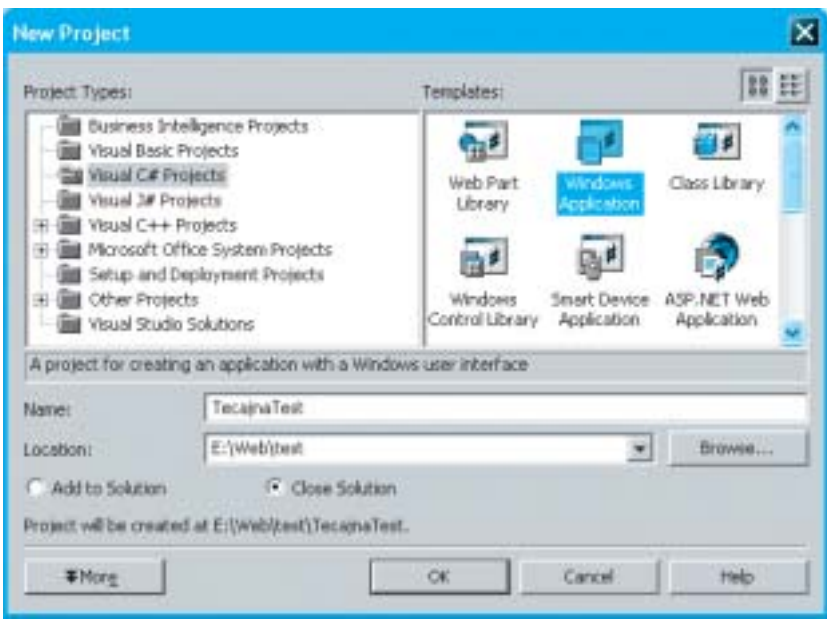
Slika 12-6: ***Rezultat poziva metode web-servisa i pretvaranja 500 USD u kune***

Web-sučelje koje je maloprije prikazano služi isključivo isprobavanju rada web-servisa – testiranju i proučavanju SOAP poruka koje se šalju te samog rada i funkcionalnosti dostupnih metoda. Prvenstvena namjena web-servisa je da budu korišteni iz drugih programa, dakle ne direktno od stvarnih korisnika – stoga će naš sljedeći korak u radu s web-servisima biti njihovo korištenje iz drugih programa pisanih u .NET-u.

Korištenje web-servisa

Iako je za korištenje web-servisa iz aplikacija nužna poveća količina kôda, vi je, kao što ste već vjerojatno i navikli, nećete morati pisati. To će za vas obaviti Visual Studio .NET. U narednom primjeru prikazat ćemo korištenje web-servisa iz prozorske aplikacije za Windowse, no rad s web-servisima se nimalo ne razlikuje u slučaju da ih odlučite upotrebljavati iz web-aplikacija (ili drugih tipova aplikacija, pa čak i iz drugih web-servisa).

Stoga započnimo izradu male aplikacije koja će služiti kao pretvarač valuta. Kako smo se u početku pri izradi web-servisa ograničili samo na jednu metodu, tako će i naša aplikacija imati malu funkcionalnost pretvaranja iz stranih valuta u kune. Stvorite stoga novi projekt kao na slici 12-7.



Slika 12-7: ***Stvaranje nove prozorske aplikacije u kojoj ćemo koristiti web-servis***

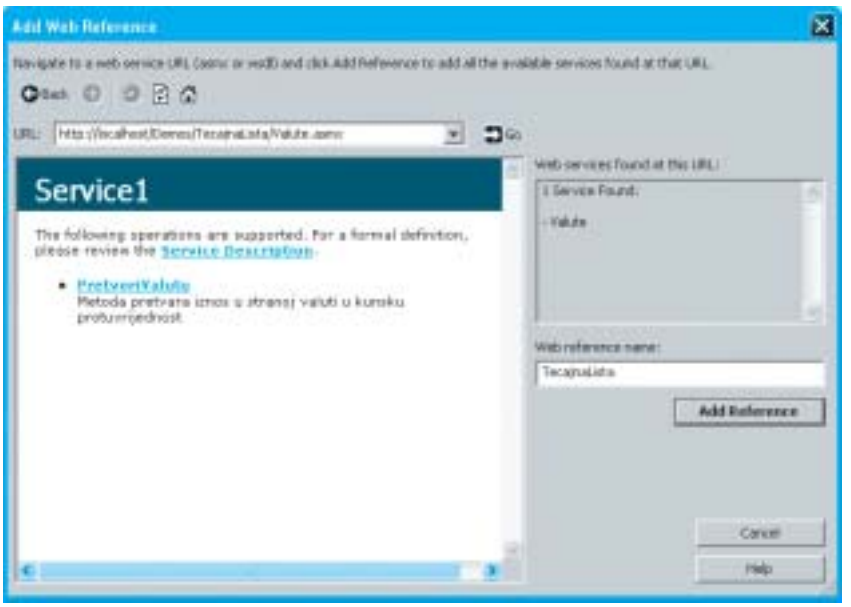
Važno je primijetiti da će sva funkcionalnost aplikacije biti sadržana u već napravljenom web-servisu, a sama prozorska aplikacija služit će isključivo kao sučelje. To je i cilj korištenja web-servisa – da sadržavaju neku programsku funkcionalnost koju je moguće pozivati iz različitih tipova programa putem weba. Ako biste željeli sad napraviti web-aplikaciju za pretvaranje valuta, napravili biste samo sučelje i pozivali web-servis za dohvat rezultata.

Dodavanje web-reference

Da bismo iz aplikacije mogli koristiti web-servise, trebamo napraviti tzv. *proxy*-klasu koja će komunicirati sa samim servisom i njegovim metodama. Kasnije u poglavlju ćemo pokazati kako se to radi ručno pomoću alata wsdl.exe, no za početak ćemo iskoristiti pomoć koju nam nudi Visual Studio .NET.

Stoga se prebacite u prozor *Solution Explorer* netom stvorene prozorske aplikacije i kliknite desnom tipkom miša na stavku *References* te odaberite opciju *Add Web Reference* i pojavit će vam se prozor kao na slici 12-8.

U prozoru Add Web Reference ne trebate ručno upisivati adresu web-servisa ukoliko je on smješten na istom računalu – jednostavno možete kliknuti na link *Browse to – Web services on the local machine* i dobit ćete popis svih web-servisa na računalu.

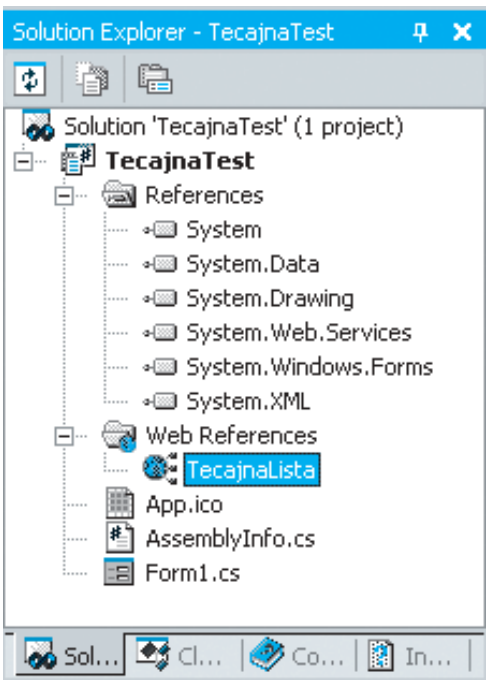


Slika 12-8: ***Dodavanje postojećeg web-servisa kao reference u aplikaciju***

U polje za URL upišite adresu web-servisa – ukoliko se on nalazi na nekom drugom računalu na Internetu, upišite njegovu punu adresu, ime servera i put do servisa, a ako se nalazi na lokalnom računalu (kao u našem primjeru), upišite adresu koja počinje s *http:///localhost/*.

Obavezno promijenite ime reference u tekstualnom polju označenom s "Web reference name". Naime, tamo će po *defaultu* stajati ime "localhost", no to nije neko odgovarajuće ime za web-servis koje želite koristiti iz svoje aplikacije. Primjerice, možete mu dati ime "TecajnaLista" jer ćete se tako lakše snalaziti.

Nakon što ste dodali web-referencu, u *Solution Exploreru* možete pronaći web-servis po stavkom *Web References*. Klikom na njega možete mu u prozoru *Properties* mijenjati osnovne postavke.



Slika 12-9:
Referenca na web-servis u Solution Exploreru

U stvarnim ćete situacijama web-servise koristiti za pružanje neke funkcionalnosti preko Interneta. Ukoliko pak želite tu funkcionalnost omogućiti samo aplikacijama na istom računalu, korištenje web-servisa nije baš efikasno jer je ipak pozivanje web-servisa sporije nego instanciranje klase s kôdom na istom računalu.

Proxy-klasa

Nakon dodavanja reference na web-servis, s njim Za to je najzaslužnija *proxy*-klasa koja je automatski stvorena stvaranjem reference na web-servis. Pogledate li u direktorij u kojem je smještena vaša aplikacija koja koristi web-servis, pronaći ćete direktorij imena "Web References". U njemu će se pak nalaziti još jedan direktorij imena dodane reference (u našem slučaju "TecajnaLista").

U njemu se, među ostalima, nalazi WSDL datoteka koja opisuje web-servis te za ovaj slučaj mnogo važnija C# datoteka (s ekstenzijom .cs) koja sadrži kôd *proxy*-klase. Ta *proxy*-klasa omogućava korištenje web-servisa iz vaše aplikacije. Slijedi ogledni sadržaj datoteke s *proxy*-klasom imena "Reference.cs".

```
//------------------------------------------------------------------------------// <autogenerated>
//   This code was generated by a tool.
//   Runtime Version: 1.1.4322.573
//
//   Changes to this file may cause incorrect behavior and will be lost if
//   the code is regenerated.
// </autogenerated>
//------------------------------------------------------------------------------

//
// This source code was auto-generated by Microsoft.VSDesigner
// Version 1.1.4322.573.

namespace TecajnaTest.TecajnaLista {
  using System.Diagnostics;
  using System.Xml.Serialization;
  using System;
  using System.Web.Services.Protocols;
  using System.ComponentModel;
  using System.Web.Services;


  /// <remarks/>
  [System.Diagnostics.DebuggerStepThroughAttribute()]
  [System.ComponentModel.DesignerCategoryAttribute("code")]

  [System.Web.Services.WebServiceBindingAttribute(Name="Service1Soap",
     Namespace="http://NET/Poglavlje12/TecajnaLista")]
  public class Service1 : System.Web.Services.Protocols.SoapHttpClientProtocol {
```

```
    /// <remarks/>
    public Service1() {
      this.Url = "http://localhost/Demos/TecajnaLista/Valute.asmx";
    }

    /// <remarks/>

[System.Web.Services.Protocols.SoapDocumentMethodAttribute("http://NET/Poglavlje12/T
ecajnaLista/PretvoriValutu", RequestNamespace="http://NET/Poglavlje12/TecajnaLista",
ResponseNamespace="http://NET/Poglavlje12/TecajnaLista",
Use=System.Web.Services.Description.SoapBindingUse.Literal,
ParameterStyle=System.Web.Services.Protocols.SoapParameterStyle.Wrapped)]
    public System.Double PretvoriValutu(System.Double Kolicina, string Valuta) {
      object[] results = this.Invoke("PretvoriValutu", new object[] {
        Kolicina,
        Valuta});
      return ((System.Double)(results[0]));
    }

    /// <remarks/>
    public System.IAsyncResult BeginPretvoriValutu(System.Double Kolicina, string
       Valuta, System.AsyncCallback callback, object asyncState) {
      return this.BeginInvoke("PretvoriValutu", new object[] {
        Kolicina,
        Valuta}, callback, asyncState);
    }

    /// <remarks/>
    public System.Double EndPretvoriValutu(System.IAsyncResult asyncResult) {
      object[] results = this.EndInvoke(asyncResult);
      return ((System.Double)(results[0]));
    }
  }
}
```

Dakle, radi se o klasi pisanoj u C# koja sadržava kôd za komunikaciju s web-servisom. Tu se nalaze sve metode za pozivanje pravih metoda web-servisa. Bacite li pogled na metodu *PretvoriValutu*, primijetit ćete da se pri njenom korištenju poziva metoda *Invoke* koja vraća polje objekata, a prvi element tog polja se pretvara u *Double* tip (koji i vraća sama metoda) te vraća kao rezultat.

Prethodni kôd se automatski generira te ga nije preporučljivo mijenjati jer će se pri sljedećem generiranju *proxy*-klase izgubiti sve promjene.

Komunikacija s web-servisom

Da biste mogli komunicirati s web-servisom, koristit ćete *proxy*-klasu preko imena web-reference, što ćete najbolje vidjeti na primjeru. Kako smo na početku stvorili prozorsku aplikaciju, dodajmo joj par kontrola koje će nam poslužiti za komunikaciju s web-servisom.

Dodajte jedno tekstualno polje koje će služiti za unos količine valute (imena *textBox1*) te jednu kontrolu *ListBox* koja će ponuditi na izbor sve dostupne valute (imena *listBox1*). Označite potom *ListBox* i u prozoru *Properties* pod *Data* – *Items* unesite kolekciju elemenata (preporučljivo je da upišete samo one s kojima web-servis zna raditi – "EUR", "USD", "CHF", "GBP").

Trebat će vam i jedan gumb koji će pokretati akciju i vraćati rezultat. Ostavite mu ponuđeno ime *button1* i smjestite ga na radnu površinu. Konačni rezultat prozora može izgledati poput prikazanog na slici 12-10.

Slika 12-10: ***Razmještaj kontrola u jednostavnoj aplikaciji za komunikaciju s web-servisom***

Još samo trebate dodati kôd koji će komunicirati s web-servisom. Da biste započeli komunikaciju s web-servisom trebat ćete samo stvoriti novi objekt maloprije objašnjene *proxy*-klase, primjerice:

```
TecajnaLista.Service1 tl = new TecajnaLista.Service1();
```

Dakle, instancirali smo objekt tipa *TecajnaLista.Service1*, što odgovara našem web-servisu – njemu smo dali ime reference *TecajnaLista*, a unutar samog web-servisa postoji klasa *Service1* u kojoj se nalaze metode za rad. Pogledajte i uvjerit ćete se da je *proxy*-klasa baš to – klasa imena *Service1* i *namespacea TecajnaLista*.

Želite li u svom kodu komunicirati s web-servisom, IntelliSense će vam ponuditi izbor svih metoda web-servisa i drugih članova za rad, kao na slici 12-11.

Slika 12-11:
IntelliSense vam nudi izbor metoda za komunikaciju s web-servisom.

Pouzdajući se u IntelliSense doista je lagano napisati kôd koji će pozvati metodu web-servisa i vratiti rezultat. Kao što vidite, sve se svodi na pozivanje neke metode iz *proxy*-klase, prosljeđivanje odgovarajućih parametara i dobivanje rezultata kao odgovora. Potpuno isto kao da radite s bilo kojom drugom klasom!

Dvaput kliknite na gumb na formularu i možete započeti s upisivanjem kôda koji će se izvršiti pri kliku na gumb. U tom trenutku želimo web-servisu proslijediti korisnikov izbor iz aplikacije (upisanu količinu valute i odabranu valutu), te na kraju ispisati rezultat.

```
private void button1_Click(object sender, System.EventArgs e)
{
    string Valuta;
    double Kolicina;

    try
    {
```

```
            Valuta = listBox1.SelectedItem.ToString();
            Kolicina = Convert.ToDouble(textBox1.Text);
        }
        catch (Exception ex)
        {
            MessageBox.Show(ex.Message, "Neispravni podaci");
            return;
        }

        TecajnaLista.Service1 tl = new TecajnaLista.Service1();
        double Rezultat = tl.PretvoriValutu(Kolicina, Valuta);

        MessageBox.Show(Rezultat.ToString() + " kn", "Rezultat");
    }
```

U prethodnom kodu koristili smo vrlo slab i općenit mehanizam za upravljanje iznimkama – njime smo htjeli spriječiti situaciju u kojoj se upisana količina valute ne može pretvoriti u broj (primjerice, korisnik je upisao znakovni niz) te situaciju u kojoj korisnik nije iz ponuđene liste odabrao valutu. Ispravno bi bilo hvatati svaki tip iznimke i ispisati odgovarajuću poruku ili pak provjeravati ulazne podatke *if*-naredbama.

U svakom slučaju, osigurajte se da u klijentskoj aplikaciji provjerite sve ulazne podatke prije njihova slanja web-servisu. Ukoliko to ne napravite, uvijek možete prepustiti web-servisu provjeravanje ulaznih podataka i vraćanje informacije o grešci, no u tom slučaju nepotrebno trošite vrijeme na komunikaciju s web-servisom. Sve provjere nad podacima obavezno napravite unutar same aplikacije, a što manje posla ostavite web-servisu.

Slika 12-1:
Aplikacija za komunikaciju s web-servisom u akciji

Metoda koja se poziva na klik gumba krajnje je jednostavna – veći njen dio zauzima kôd za obradu podataka iz formulara, a tek dvije naredbe služe za komunikaciju s web-servisom. Prvo se instancira novi objekt *tl* tipa *TecajnaLista.Service1*, a zatim se taj isti objekt koristi za pozivanje metode *PretvoriValutu* s dva parametra. Rezultat se vraća i sprema u varijablu *Rezultat* koja je tipa *double*.

Na samom kraju ispisujemo poruku s rezultatom u malom prozoru, kao na slici 12-13.

Slika 12-13:
Rezultat poziva web-servisa ispisan naredbom MessageBox.Show

Asinkrono pozivanje web-servisa

Vratite li se natrag u kôd *proxy*-klase, vidjet ćete da osim metode *PretvoriValutu* postoje još dvije metode sličnog naziva, *BeginPretvoriValutu* i *EndPretvoriValutu*. One služe za tzv. asinkrono pozivanje web-servisa, dok metoda *PretvoriValutu* služi za sinkrono pozivanje.

U prethodnom primjeru smo metodu web-servisa pozivali sinkrono. Da pojasnimo, *sinkrono* znači da smo pozvali metodu, a aplikacija se zatim pauzirala i čekala na odgovor web-servisa. Tek kad je on stigao, nastavilo se s izvršavanjem sljedeće naredbe. Asinkroni poziv omogućava suprotno – pozovemo li metodu asinkronim načinom, naša aplikacija će moći nastaviti s radom, a kad stigne odgovor web-servisa, izvršit će se određena akcija. Time štedimo vrijeme klijentske aplikacije koja poziva web-servis – ukoliko je njegovo izvršavanje dugotrajno (bilo zbog kompliciranog zadatka koji mora obaviti ili zagušene internetske veze), aplikacija ne mora čekati već može nastaviti s izvršavanjem.

Iako je naš web-servis brz jer je njegov zadatak lagan (ne otvara bazu podataka, ne obavlja složene izračune), a i nalazi se na lokalnom stroju, što dodatno ubrzava komunikaciju, na njegovu primjeru ćemo pokazati asinkrono pozivanje.

Pojednostavljena verzija asinkronog poziva uključuje korištenje metoda *Begin* i *End* (njihov sufiks odgovara nazivu metode web-servisa). Tako ćemo pozvati *BeginPretvoriValutu* te time proslijediti

parametre web-servisu i zadati mu posao, no u aplikaciji ćemo nastaviti s izvršavanjem drugog kôda i kasnije tek završiti komunikaciju s web-servisom pozivanjem metode *EndPretvoriValutu*.

Slijedi izmijenjena verzija dijela prethodno prikazanog kôda:

```
TecajnaLista.Service1 tl = new TecajnaLista.Service1();
IAsyncResult rez = tl.BeginPretvoriValutu(Kolicina, Valuta, null, null);

// nastavljamo s izvršavanjem aplikacije

double Rezultat = tl.EndPretvoriValutu(rez);
```

Dakle, prvo pozivamo *BeginPretvoriValutu* koja, uz parametre metode *PretvoriValutu*, prima još dva parametra koje ćemo koristiti kasnije, a zasad ih postavljamo na vrijednosti *null*. Time zadajemo zadatak web-servisu i ostavljamo ga da obradi parametre i dostavi odgovor i rezultat.

Za asinkronu komunikaciju s web-servisom koristi se sučelje *IAsyncResult* koje služi za spremanje rezultata asinkronog poziva. To sučelje nije vezano isključivo uz rad s web-servisima, već asinkrono možete raditi i s drugim resursima u .NET-u. Pri pozivu naredbe *EndPretvoriValutu* prosljeđuje se taj objekt za asinkronu komunikaciju.

Ukoliko web-servis ne dostavi rezultat do trenutka kad se poziva *EndPretvoriValutu*, blokira se daljnje izvršavanje aplikacije sve dok ne stigne odgovor. Dakle, ista situacija kao kad se koristi sinkrono pozivanje web-servisa, samo što između *Begin* i *End* naredbe imate priliku izvršiti dio kôda u klijentskoj aplikaciji.

Možete iskoristiti i *IsCompleted* svojstvo objekta za asinkronu komunikaciju za provjeru je li poziv web-servisu završen i je li stigao odgovor. Tako možete izbjeći prerano pozivanje naredbe *End* i time blokiranje daljnjeg izvršavanja aplikacije.

```
double Rezultat;
if (rez.IsCompleted)
{
   Rezultat = tl.EndPretvoriValutu(rez);
}
else
{
   // izvrši još kôda i pokušaj ponovno kasnije
}
```

Drugi način izvršavanja asinkronih upita web-servisu je korištenje *callback* delegata odnosno pokazivača na metodu koja će se izvršiti kad se vrati rezultat asinkronog poziva. To je ujedno i najpraktičniji način izvršavanja asinkronih poziva jer se svodi na sljedeći scenarij: vi zadate upit web-servisu i

pokazivač na metodu koju treba izvršiti u trenutku kad web-servis vrati odgovor, a vaša aplikacija ostaje potpuno slobodna za daljnje izvršavanje kôda.

Za to će vam trebati pokazivač na metodu koja će se pozvati po primitku rezultata web-servisa. Evo kako bi izgledao kôd s korištenjem pokazivača na *callback* metodu:

```
TecajnaLista.Service1 tl;

private void button1_Click(object sender, System.EventArgs e)
{
   // isti kôd kao i u originalnom primjeru

   tl = new TecajnaLista.Service1();

   AsyncCallback povratnaMetoda = new AsyncCallback(RezultatStigao);

   tl.BeginPretvoriValutu(Kolicina, Valuta, povratnaMetoda, null);
}

private void RezultatStigao(IAsyncResult rez)
{
   double Rezultat = tl.EndPretvoriValutu(rez);
   MessageBox.Show(Rezultat.ToString() + " kn", "Rezultat");
}
```

U metodi koja se izvršava na klik gumba u aplikaciji je izostavljen početni dio kôda, koji je isti kao i u prethodnim primjerima (provjeravanje sadržaja kontrola i postavljanje vrijednosti varijabli *Kolicina* i *Valuta*).

Dakle, stvaramo delegat imena *povratnaMetoda* tipa *AsyncCallback* (kao što i ime kaže, radi se o *callback* metodi za asinkrone pozive). Pri njegovu stvaranju kao parametar koristimo ime metode koja će se pozvati kad stigne odgovor od web-servisa, intuitivnog imena *RezultatStigao*.

Potom samo pozivamo metodu *BeginPretvoriValutu*, a kao treći parametar prosljeđujemo netom stvoreni *callback* delegat imena *povratnaMetoda*.

Sama *callback* metoda za parametar mora obavezno primati objekt tipa *IAsyncResult*, jer će u njemu biti spremljene informacije o rezultatu asinkronog poziva. U metodi nastavljamo poznatim kôdom – pozivom metode *EndPretvoriValutu* i objektom rezultata asinkronog poziva kao parametrom dobivamo rezultat poziva web-metode koji prikazujemo s *MessageBox.Show*.

U gornjem primjeru poziv *EndPretvoriValutu* neće blokirati daljnji rad aplikacije, već će se izvršiti odmah jer se nalazi u metodi koja se poziva tek u trenutku kad stigne rezultat web-servisa.

Iako je naš servis dosta brz te možda nije odmah vidljiva isplativost korištenja asinkronih poziva, nju je svakako korisno implementirati pri pozivanju web-servisa koji obavljaju složene operacije i onih smještenih na udaljenim poslužiteljima, jer su realne mogućnosti zagušenja mreže, što ne možete kontrolirati ni predvidjeti.

Komandno-linijski alat wsdl.exe

Ključni korak u izgradnji aplikacije koja komunicira s web-servisom je izrada *proxy*-klase. I dok će Visual Studio .NET to automatski napraviti za vas u trenutku kad dodajete web-referencu svom projektu, dobro je znati što se događa u pozadini tog postupka.

Alat wsdl.exe se nalazi u direktoriju "C:\Program Files\Microsoft Visual Studio .NET 2003\SDK\v1.1\Bin" (naravno, pod uvjetom da ste Visual Studio .NET 2003 instalirali na *defaultnu* lokaciju). No najčešće ćete željeti wsdl.exe pozivati iz nekog drugog direktorija, vjerojatno iz onog u kojem je smještena aplikacija koja želi komunicirati s web-servisom. Kako ćete to raditi iz komandno-linijskog prompta (*Start – All Programs – Accessories – Command Prompt*), komplicirano bi bilo svaki put pisati punu putanju do wsdl.exe datoteke, pa je preporučljivo njen direktorij dodati u *path*. Tako ćete iz bilo kojeg direktorija moći pozivati sve datoteke iz tog direktorija samo navodeći njihovo ime. U komandno-linijskom promptu upišite sljedeću naredbu:

```
path=%path%;C:\Program Files\Microsoft Visual Studio .NET
2003\SDK\v1.1\Bin
```

Sad ćete moći iz bilo kojeg direktorija pozvati wsdl.exe bez navođenja pune putanje do njegove lokacije na disku.

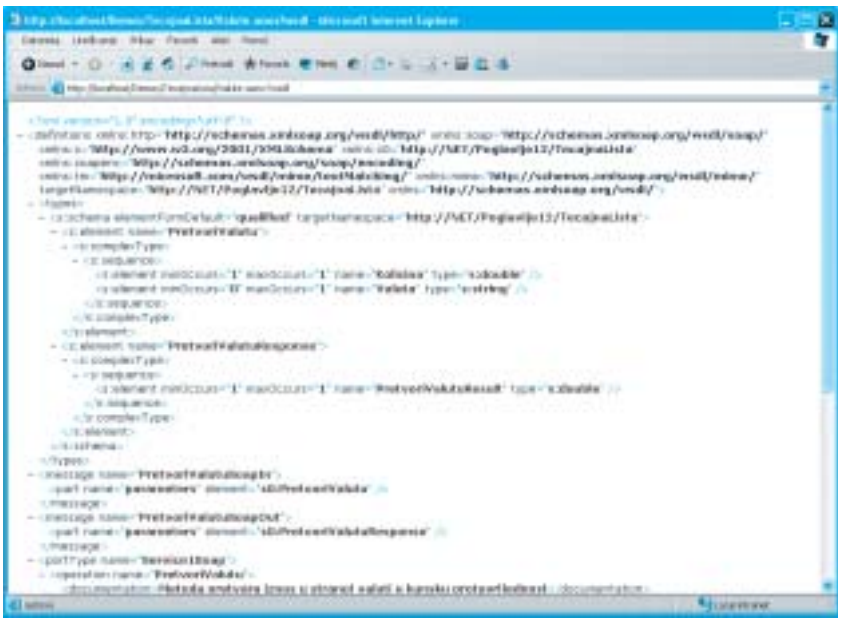
No postoji i jednostavniji način – otvorite posebnu verziju komandne linije koja se nalazi u *Start – All Programs – Visual Studio .NET 2003 – Visual Studio .NET Tools – Visual Studio .NET 2003 Command Prompt*. Radi se o podešenoj verziji komandne linije koja već ima podešen *path*, pa su vam svi alati dostupni samo preko njihova imena.

Naime, dok vi bezbrižno klikćete po Visual Studiju, u pozadini se poziva komandno-linijski alat wsdl.exe koji obavlja cijeli posao generiranja *proxy*-klase. U nastavku ćemo ručno upravljati alatom wsdl.exe, jer on nudi nekoliko zanimljivih opcija i predstavlja ključan korak u radu s web-servisima.

Korištenje wsdl.exe alata za stvaranje *proxy*-klase krajnje je jednostavno, izuzmemo li činjenicu da se radi u komandno-linijskom načinu rada, na što možda niste navikli. Želite li tako stvoriti *proxy*--klasu, za to će vam trebati WSDL opis web-servisa. Na svu sreću, uz svaki web-servis automatski se generira njegov WSDL opis tako da mu dodate sufiks "?wsdl".

Primjerice, želite li stvoriti *proxy*-klasu za web-servis koji smo izradili u ovom poglavlju, mogli biste napisati:

```
wsdl http://localhost/Demos/TecajnaLista/Valute.asmx?wsdl
```



Slika 12-14: *WSDL opis web--servisa možete vidjeti i u pregledniku upišete li adresu servisa i dodate li joj "?wsdl"*

Dakle, proslijedili smo mu adresu web-servisa sa sufiksom "?wsdl", što automatski daje njegov WSDL--opis. Evo kako se odvija komunikacija s alatom wsdl.exe (prijepis rezultata iz komandno-linijskog prompta).

```
E:\Web\test\TecajnaTest>wsdl http://localhost/Demos/TecajnaLista/Valute.asmx?wsdl

Microsoft (R) Web Services Description Language Utility
```

```
[Microsoft (R) .NET Framework, Version 1.1.4322.573]
Copyright (C) Microsoft Corporation 1998-2002. All rights reserved.

Writing file 'E:\Web\test\TecajnaTest\Service1.cs'.

E:\Web\test\TecajnaTest>_
```

On je, dakle, stvorio CS datoteku istog imena kao i klasa web-servisa i zapisao ju je u direktorij iz kojeg smo ga pozvali.

Pri korištenju alata wsdl.exe ne trebate web-servisu dodati sufiks "?wsdl", već će se on, ukoliko to ne učinite, dodati automatski. Tako ste prethodni primjer mogli izvršiti naredbom:

```
wsdl http://localhost/Demos/TecajnaLista/Valute.asmx
```

No važno je znati da se za stvaranje *proxy*-klase koristi WSDL-opis web-servisa (zato se i alat za njeno generiranje zove wsdl.exe).

Ukoliko pak želite napraviti *proxy*-klasu za web-servis koji nije pisan u .NET-u, možda vam neće biti dostupan njegov WSDL opis na isti način kao u gornjim primjerima (preko sufiksa "?wsdl"). U tom slučaju ćete vjerojatno imati običnu WSDL datoteku s opisom iz koje ćete generirati *proxy*-klasu. Tako alatu možete proslijediti običnu WSDL datoteku s opisom nekog web-servisa spremljenu na lokalnom računalu – rezultat je isti.

```
wsdl mojServis.wsdl
```

Kao što ste vidjeli, tako se stvara datoteka istog imena koje ima i klasa web-servisa. To možete promijeniti i generirati CS-datoteku drugog imena. Za to služi opcija /out koju zadajete wsdl.exe alatu.

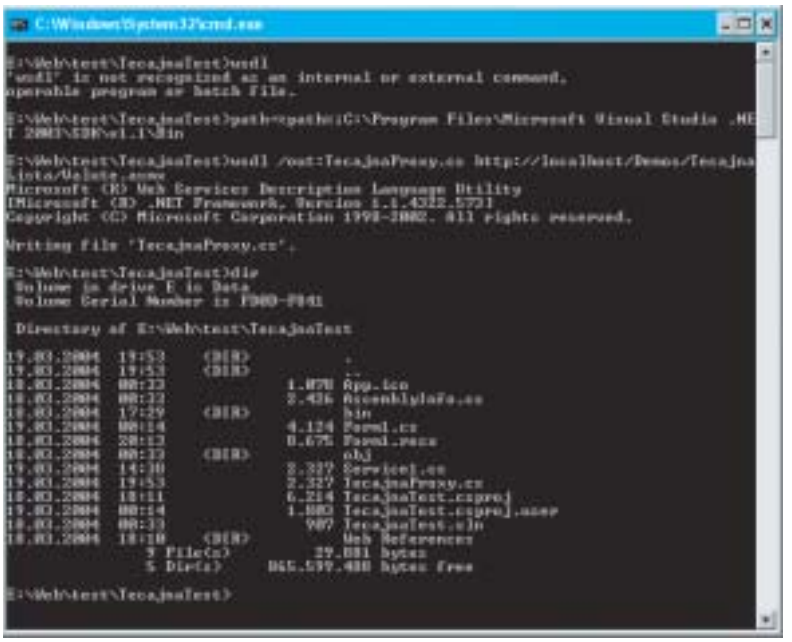
```
wsdl /out:TecajnaProxy.cs http://localhost/Demos/TecajnaLista/Valute.asmx
```

Kao što i u .NET-u imate mogućnost pisanja kôda u različitim jezicima, tako možete i *proxy*-klasu generirati u Visual Basic .NET jeziku, što vam može poslužiti kod proučavanja njenog sadržaja.

```
wsdl /language:vb http://localhost/Demos/TecajnaLista/Valute.asmx
```

Za to će vam, kao što vidite, poslužiti opcija /language, nakon koje odvojeno dvotočkom navodite jezik. Naravno, odabir jezika u kojem će biti napisana *proxy*-klasa nema nikakvog utjecaja na njen rad.

Slika 12-15: ***Primjer korištenja wsdl.exe alata – primijetite postavljanje "patha" i stvaranje proxy-klase u datoteci s drugačijim imenom.***



Želite li da kôd generirane *proxy*-klase bude u posebnom *namespaceu*, što može biti jako korisno ukoliko želite spriječiti podudaranja tipova definiranih u vašoj aplikaciji i onih u *proxy*-klasi, možete iskoristiti /namespace opciju.

```
wsdl /namespace:WSTecajna http://localhost/Demos/TecajnaLista/Valute.asmx
```

U nastavku slijedi tablica u kojoj će biti detaljnije opisane sve opcije wsdl.exe alata.

Tablica 12-1: ***Opcije alata wsdl.exe***

Opcija	Opis
/nologo	miče "pozdravnu" poruku pri pokretanju, koja sadrži ime alata i verziju
/language:<jezik>	jezik koji će biti korišten za generiranje proxy-klase; možete izabrati između "CS", "VB" ili "JS" ili ponuditi puno ime klase koja implementira *System.CodeDom.Compiler-.CodeDomProvider*, čime imate mogućnost *proxy*-klasu generirati u bilo kojem .NET jezik; *defaultna* postavka je "CS" odnosno C#; kratica za ovu opciju je /l:<jezik>.

Opcija	Opis
/server	generira apstraktnu klasu za web-servis korištenjem ASP.NET-a
/namespace:<ime>	dozvoljava generiranje *proxy*-klase s posebnim *namespaceom*; kratica za ovu opciju je /n:<ime>
/out:<datoteka>	ime datoteke pod kojom će se spremiti generirana klasa; ukoliko ne navedete ime, datoteka će dobiti ime iz klase web-servisa; kratica za ovu opciju je /o:<datoteka>
/protocol:<protokol>	korištenjem ove opcije možete prekoračiti *defaultni* SOAP protokol za komunikaciju s web-servisom i odabrati GET ili POST metode; dozvoljene opcije su "SOAP", "HttpGet" i "HttpPost"
/username:<korisničko_ime> /password:<zaporka> /domain:<domena>	postavke koje će se koristiti za spajanje na poslužitelj koji zahtijeva autentikaciju; kratice za ove opcije su /u:<korisničko_ime>, /p:<zaporka>, /d:<domena>.
/proxy:<url>	URL *proxy* servera za zadavanje HTTP zahtjeva; po *defaultu* će se koristiti sistemske postavke (Napomena: *proxy* server nema nikakve veze s *proxy*-klasom koja se generira, već je on dio postavki veze računala s Internetom.)
/proxyusername: <korisničko_ime> /proxypassword:<zaporka> /proxydomain:<domena>	postavke koje će se koristiti za spajanje preko *proxy* poslužitelja koji zahtijeva autentikaciju; kratice za ove opcije su /pu:<korisničko_ime>, /pp:<zaporka> i /pd:<domena>.
/appsettingurlkey:<ključ>	konfiguracijski ključ koji će se koristiti u kodu *proxy*-klase za čitanje postavki URL-a web-servisa; ako koristite konfiguracijske datoteke ne trebate imati URL web-servisa hardkodiran u *proxy*-klasi, već može biti smješten u konfiguracijskoj datoteci; kratica za ovu opciju je /urlkey:<ključ>.
/appsettingbaseurl: <bazni_URL>	bazni dio URL-a koji će se koristiti pri računanju adrese web-servisa (ona će biti relativna u odnosu na bazni URL definiran ovom opcijom); pri korištenju ove opcije obavezno treba koristiti i appsettingurlkey opciju; kratica za ovu opciju je /baseurl:<bazni_URL>.

Proxy-klasu generiranu wsdl.exe alatom možete kompajlirati u DLL datoteku i spremiti u direktorij *bin* unutar projekta u kojem želite komunicirati s web-servisom, a taj DLL možete referencirati korištenjem ključne riječi *using* u kodu aplikacije.

SOAP ili ne-SOAP?

Proxy-klasa koju generirate pomoću wsdl.exe alata po *defaltu* će biti izvedena iz klase *System.Web.Services.Protocols.SoapHttpClientProtocol* koja omogućava pozivanje web-servisa korištenjem SOAP-a preko HTTP protokola, što je osnovna ideja i standardni način rada web-servisa. No vi ipak niste na to ograničeni – možete promijeniti protokol kojim će biti komunicirano s web-servisom korištenjem opcije /protocol. Sljedeći primjer tako stvara *proxy*-klasu izvedenu iz *HttpGetClientProtocol*, što znači da će se komunikacija s web-servisom odvijati preko metode GET protokola HTTP.

```
wsdl /protocol:httpget
http://localhost/Demos/TecajnaLi
sta/Value.asmx
```

Na isti način ste mogli odlučiti koristiti metodu POST – kao protokol biste naveli "httppost".

Zašto se uopće odlučiti na promjenu protokola za komunikaciju s web-servisom? U većini slučajeva, SOAP je najbolje rješenje. No SOAP se temelji na XML-u i sve poruke su "upakirane" u posebne *tagove*, tzv. SOAP omotnicu (engl. *envelope*). Komunicirate li s web-servisom prosljeđujući i dobivajući natrag jednostavne tipove podataka (primjerice, brojeve), korištenjem GET ili POST metoda ti će pozivi biti malo efikasniji jer će se slati manje podataka preko mreže (neće biti okolnog XML kôda).

Ukoliko koristite Visual Studio .NET za rad s web-servisima, možete se potpuno pouzdati u njegovu *Add Web Reference* opciju za stvaranje *proxy*-klasa, jer se u pozadini ionako pokreće alat wsdl.exe. Ta opcija također može dodavati i UDDI web-servise, što vam dodatno olakšava rad. No ukoliko se nađete u situaciji da iz web-aplikacije smještene na nekom poslužitelju komunicirate s web-servisom koji se u međuvremenu malo promijenio, na tom poslužitelju najčešće nećete imati Visual Studio .NET i morat ćete se pouzdati u ručno generiranje novih *proxy*-klasa. Takav ćete način rada prakticirati i u slučajevima kad želite dodatno podesiti *proxy*-klase. No imajte na umu da ćete pri svakoj promjeni web-servisa morati ponovno generirati *proxy*-klasu – s druge strane, za takve situacije vam Visual Studio .NET pruža mnogo praktičniju opciju *Update Web Reference*.

IV. dio

Dodaci

DODATAK A: .NET COMPACT FRAMEWORK

DODATAK B: DEBUGIRANJE

DODATAK C: POMOĆNI ALATI ZA .NET

DODATAK D: DVD

DODATAK A

.NET Compact Framework

U ovom poglavlju:

- Što je .NET Compact Framework
- Kako napraviti aplikaciju za Pocket PC

Cijelo vrijeme pričamo o razvoju aplikacija za osobna računala – što je s ručnim računalima i ostalim pametnim uređajima (engl. *smart devices*)? Implementacija punog .NET Frameworka, zbog njihovih brojnih ograničenja, ne bi bila dobro rješenje. Stoga je napravljen .NET Compact Framework – srezana inačica .NET Frameworka prilagođena mogućnostima tih uređaja.

Slično kao kod osobnih računala, da bi se aplikacije pisane na platformi .NET mogle koristiti, mora postojati podrška, bilo kao naknadna instalacija, bilo kao sastavni dio operativnog sustava. .NET Compact Framework (skraćeno .NET CF) u trenutku pisanja ovih redaka dio je operativnog sustava Windows CE .NET 4.2, a izvjesno je da će tako biti i sa svim sljedećim verzijama. Kako je sustav za ručna računala Windows Mobile 2003 (ponegdje referenciran kao Pocket PC 2003) baziran na spomenutoj verziji Windowsa CE, i on u sebi ima .NET CF.

Na sustave bazirane na Windows CE .NET 4.1 .NET Compact Framework može se instalirati u vidu dodatne instalacije, a isto vrijedi i za neke sustave bazirane na Windowsima CE 3.0, konkretno Pocket PC 2000 i Pocket PC 2002.

Međutim, čak i ako ciljani operativni sustav sadrži .NET CF, preporuka je nadograditi postojeće stanje aktualnim *service packom*. Adresa za preuzimanje nadogradnji i kompletnih inačica je:

```
http://msdn.microsoft.com/mobility/downloads/updates/
```

Ukoliko imate Visual Studio .NET 2003 Professional (ili bolji) .NET Compact Framework će doći s njim i po potrebi biti automatski instaliran na uređaj pri prvom pokretanju mobilne aplikacije. Više o tome kasnije u poglavlju.

Kao što ste vjerojatno očekivali, i .NET Compact Framework je usko povezan s razvojnim alatom Visual Studio. Međutim, on brojčano nešto sporije napreduje, tako da se uz Visual Studio .NET 2003 koristi .NET Compact Framework inačice 1.0, a aktualna verzija u trenutku pisanja označena je kao SP2. U njoj su popravljene i unaprijeđene određene mogućnosti, no osnovne karakteristike su ostale iste.

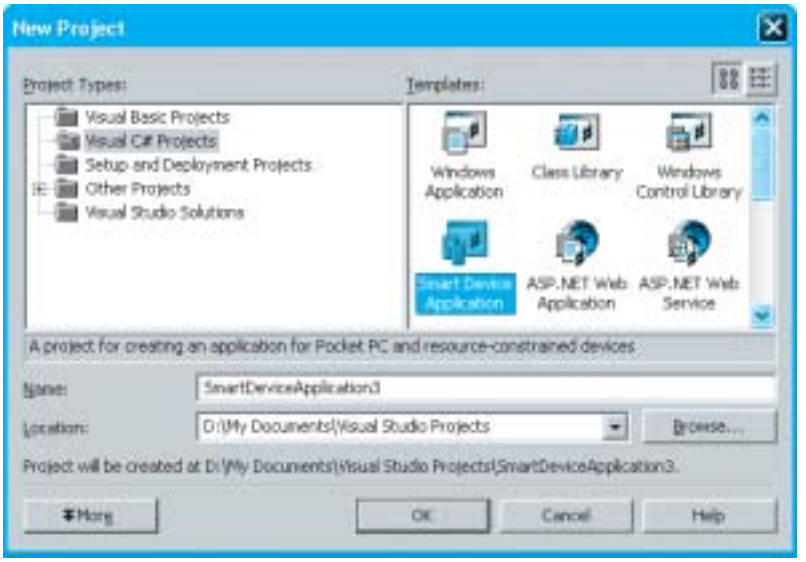
.NET Compact Framework podržava programiranje isključivo u jezicima VB.NET i C#.

Ukoliko nemate pri ruci ručno računalo, to nije razlog da ne pišete aplikacije za njega. Naime, Visual Studio ima ugrađen emulator ručnog računala, pa vam prilikom razvoja ono uopće nije potrebno. Ipak, planirate li se ozbiljno time baviti, svakako nabavite i uređaj – sasvim je drugačiji osjećaj koristiti aplikaciju u emulatoru ili tapkajući olovčicom po pravoj stvari.

U ovom ćemo dodatku zajedno napraviti jednostavnu aplikaciju koristeći .NET Compact Framework te tako ukratko upoznati ograničenja i specifičnosti takvih aplikacija.

Izrada aplikacije za Pocket PC

Kao i svaku drugu aplikaciju, onu za .NET Compact Framework počinjemo stvaranjem novog projekta. Ovoga puta treba odabrati predložak Smart Device Application, kao što možete vidjeti na slici A-1.



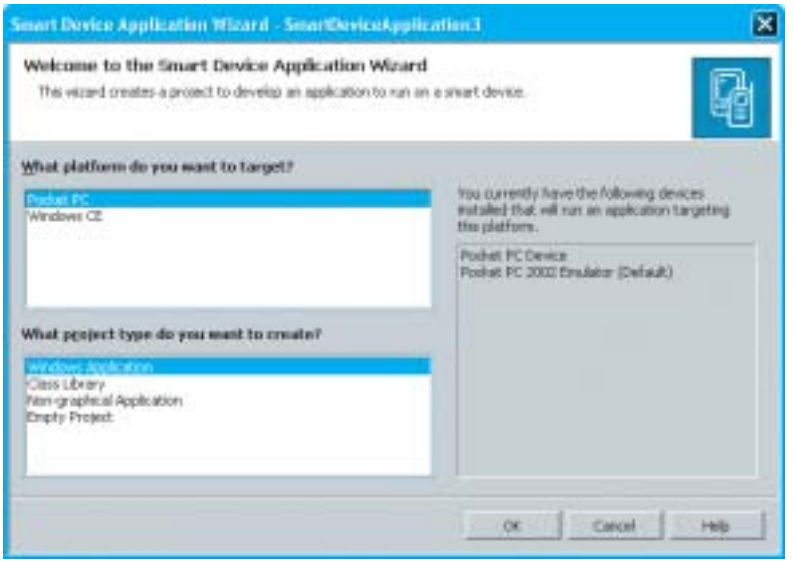
Slika A-1: ***Stvaranje novog projekta za pametne uređaje***

Ukoliko u popisu nema potrebnog predloška, pokrenite instalaciju Visual Studija, potražite stavku Smart Device Programmability (unutar stavke Visual C# .NET) i uključite je.

Nakon odabira predloška i unošenja imena, aplikacija pojavit će se dodatni prozor koji će nam pomoći pri postavljanju osnovnih parametara (slika A-2).

Ako instalirate Smartphone SDK, pojavit će se i predložak za pisanje aplikacija za Smartphone.

Prva, izuzetno važna odluka je odluka o ciljanoj platformi. Ukoliko izaberete platformu Windows CE, onda će se vaša aplikacija moći izvršavati na svim uređajima koji koriste taj operativni sustav. Kao što smo već spomenuli, sustavi Pocket PC 2000, Pocket PC 2002 i Windows Mobile 2003 koje nalazimo u ručnim računalima Pocket PC baziraju se na toj platformi, što znači da ćete aplikaciju moći koristiti na njima, ali i brojnim drugim uređajima koji koriste tu platformu.



Slika A-2: ***Pomoćni prozor za postavljanje parametara aplikacije***

Detaljan opis razlika između razvoja aplikacija za Pocket PC i Windows CE općenito potražite na adresi:

`http://msdn.microsoft.com/library/enus/dnnetcomp/html/netcfPPCtoCE.asp`

Međutim, mi ćemo za naš primjer izabrati platformu Pocket PC te se tako limitirati isključivo na sustave Pocket PC 2000, Pocket PC 2002 i Windows Mobile 2003. Zauzvrat će Visual Studio prilagoditi sve potrebne parametre kako bi razvoj takve aplikacije bio što jednostavniji. Naime, uređaji koji koriste Windows CE variraju po brojnim karakteristikama, pa je razvoj "općenite aplikacije" toliko kompliciran da je specijalizacija puno elegantnije rješenje. Ionako ćete u većini slučajeva razvijati aplikaciju upravo za uređaje tipa Pocket PC.

U drugom dijelu prozora biramo tip aplikacije koji želimo napraviti – mi smo se odlučili za tip Windows Application, a vi, ovisno o željama i potrebama, slobodno eksperimentirajte i s ostalim tipovima.

Forme i kontrole

Sučelje koje će nas dočekati nakon potvrde ovih izbora gotovo je identično početnom sučelju za izradu prozorskih aplikacija u .NET Frameworku. Zapravo, jedina je razlika broj dostupnih kontrola i njihovih svojstava te izgled određenih kontrola.

(Pri)ručna baza na ručnom računalu

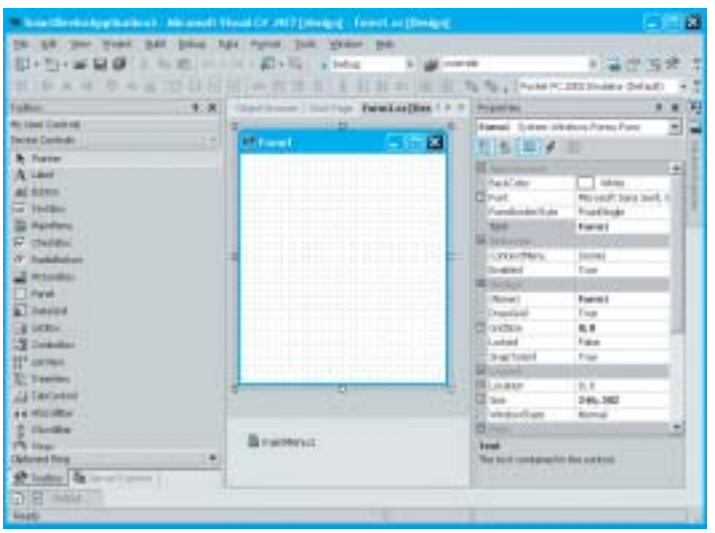
Radite li na pametnom uređaju aplikaciju koja zahtijeva rad s bazom podataka, mogli biste pomisliti da se nalazite pred nepremostivom zaprekom. Srećom, tu je proizvod SQL Server CE, znatno pojednostavljena i optimizirana inačica velike poslužiteljske baze SQL Servera 2000.

Taj proizvod, koji besplatno možete preuzeti s adrese http://www.microsoft.com/sql/ce/, može raditi na dva načina – kao samostalna baza podataka ili kao svojevrstan međuspremnik podataka koji se povremeno sinkronizira s centralnom, velikom bazom.

Iako sama baza podataka nije sastavni dio .NET Compact Frameworka, međusobno vrlo dobro surađuju. Tako podršku za SQL Server CE možemo pronaći u Visual Studiju .NET 2003, a podacima iz aplikacije pristupamo koristeći klase u biblioteci ADO.NET. Naravno, iz očitih razloga, postoji velik broj stvari koje nisu podržane u odnosu na SQL Server za osobna računala.

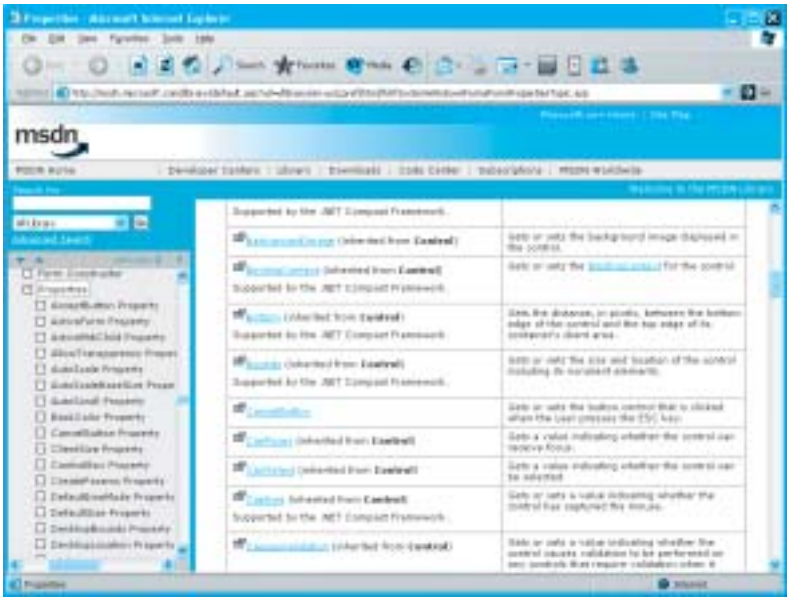
Naravno, bazama smještenim na poslužitelju iz aplikacija na ručnom računalu možete pristupati direktno, no u tom slučaju među njima mora postojati stalna veza, što zbog naravi pametnih uređaja često nije slučaj.

Za rad sa SQL Serverom CE koristi se namespace System.Data.SqlServerCE, a za direktan rad s poslužiteljskom bazom System.Data.SqlClient.



Slika A-3: ***Osnovno sučelje za razvoj aplikacija za Pocket PC***

Vjerojatno ste prilikom proučavanja dokumentacije primijetili kako određene kontrole, svojstva, metode i događaji imaju napomenu *Supported by the .NET Compact Framework*. One koji imaju tu napomenu podržava .NET Compact Framework. Naravno, za to ne morate svako malo viriti u dokumentaciju – izborom predloška Smart Device Application Visual Studio automatski skriva sve nepodržane karakteristike i mogućnosti, kako u pomoćnim prozorima tako i u tehnologijama poput IntelliSensa.



Slika A-4: ***Dokumentacija ima jasno naznačeno što podržava u .NET Compact Frameworku***

Prilikom otvaranja nove forme automatski će biti dodana kontrola za izbornik. To je stoga što bi svaka aplikacija za Pocket PC trebala imati izbornik. Vi ga, dakako, možete ukloniti, no time nećete uštedjeti prostor jer se izbornik na ručnim računalima nalazi u sistemskoj traci.

Poznato je da ručna računala slijede logiku nešto drugačiju od one na osobnim računalima. Jedna od glavnih razlika je da programi na ručnom računalu klikom na iksić ne bivaju zatvoreni, nego se minimiziraju. Takvo ponašanje posebno je nezgodno za vrijeme razvoja aplikacije jer će *debugger* ostati aktivan sve dok aplikaciju ne zatvorimo. Stoga prilikom kreiranja nove forme najprije treba postaviti svojstva MinimizeBox na *false*. Na taj način onemogućit ćemo minimiziranje aplikacije, a umjesto iksića pojavit će se gumbić s natpisom "ok". Klik na taj gumbić zatvorit će formu, a ukoliko je osnovna, i aplikaciju.

Mi ćemo za primjer napraviti jednostavnu aplikaciju za prevođenje s engleskog jezika na talijanski, i obrnuto. Koje smo kontrole koristili možete vidjeti na slici A-4 – uz kozmetičke, nužno je postaviti sljedeće: textUnos (tip TextBox), buttonEnIt i buttonItEn (tip Button) te labelPrijevod (tip Label).

Slika A-3:
Osnovno sučelje za razvoj aplikacija za Pocket PC

Za sam čin prevođenja koristimo besplatan web-servis, koji se nalazi na adresi

```
http://www.webservicex.com/TranslateService.asmx
```

Postupak dodavanja Web Reference u projekt znate iz prošlog poglavlja – desni klik na ime projekta u Solution Exploreru, zatim Add Web Reference gdje unosimo sljedeću adresu:

```
http://www.webservicex.com/TranslateService.asmx?WSDL
```

Kôd koji će odraditi funkcionalnost programa izgleda ovako – sljedeća funkcija vezana je uz oba gumba na našoj formi:

```
private void button_Click(object sender, System.EventArgs e)
{
  com.webservicex.www.Language jezik;
```

```
  if ((Button)sender == buttonEnIt)
    jezik = com.webservicex.www.Language.EnglishTOItalian;
  else
    jezik = com.webservicex.www.Language.ItalianTOEnglish;

  com.webservicex.www.TranslationService ws = new
            com.webservicex.www.TranslationService();
  ws.Timeout = 20000; // na odgovor čekamo do 20 sekundi

  try
  {
    labelPrijevod.Text = ws.Translate(jezik, textUnos.Text);
  }
  catch
  {
    labelPrijevod.Text = "Web-servis nije dostupan.";
  }
}
```

Virtualna tipkovnica

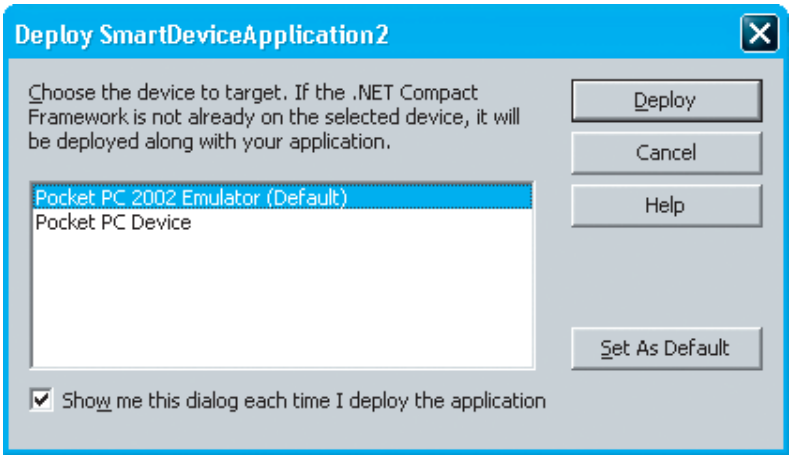
Sve dosad rečeno nije ništa novo. Međutim, ako pokrenete aplikaciju (na ručnom računalu, ne u emulatoru), primijetit ćete da ručno morate dozvati tipkovnicu kada želite upisati riječ. Nije da je to nešto strašno, no korisnici bi cijenili kada bismo napravili da se virtualna tipkovnica sama uključuje prilikom ulaska u polje za upis i isključuje izlaskom iz njega.

Stoga kraj forme treba dovući kontrolu InputPanel, a uz događaje GetFocus i LostFocus kontrole textUnos vezati sljedeće funkcije:

```
private void textUnos_GotFocus(object sender, System.EventArgs e)
{
  inputPanel1.Enabled = true;
}

private void textUnos_LostFocus(object sender, System.EventArgs e)
{
  inputPanel1.Enabled = false;
}
```

Testiranje aplikacije

Preostaje nam još testirati aplikaciju koju smo upravo napravili. I tu je priča vrlo slična izradi ostalih tipova aplikacija – jednostavno pritisnemo F5 i kreće postupak kompajliranja i pokretanja aplikacije.



Slika A-6: ***Izbor uređaja na kojem ćemo testirati aplikaciju***

Kako aplikacija koju smo upravo napisali nije namijenjena izvršavanju na osobnom računalu, morat ćemo koristiti emulator računala Pocket PC ili koristiti pravo ručno računalo. Zato će nam Visual Studio prilikom pokretanja programa postaviti pitanje sa slike A-6.

U pravilu je praktičnije testiranje raditi na emulatoru, pa stoga odabiremo njegovu stavku među ponuđenim opcijama. Kako sada prvi put pokrećemo emulator, na njega će prije pokretanja aplikacija automatski biti instaliran .NET Compact Framework.

Ista stvar bi se dogodila i na (pravom) ručnom računalu da smo njega izabrali za *deployment* aplikacije. Naravno, u trenutku pokretanja ručno računalo mora biti spojeno na osobno računalo, a ActiveSync, softver za njihovu sinkronizaciju i povezivanje, pravilno podešen i pokrenut.

Kao i kod ostalih aplikacija pisanih za platformu .NET, i ovdje je dovoljno kopirati izvršnu datoteku na ciljni uređaj i stvar će raditi. Želite li pak instalacijsku proceduru, morat ćete se pomučiti nešto više, kako je opisano na adresi http://msdn.microsoft.com/library/en-us/dnnetcomp/html/-netcfdeployment.asp.

Slika A-7:
Prilikom prvog pokretanja aplikacije iz Visual Studija automatski će se na odabrani uređaj instalirati nova inačica .NET Compact Frameworka.

Kako dalje?

Na ovih nekoliko stranica spomenuli smo samo osnove izrade aplikacija za pametne uređaje. Kako se cijela priča uvelike oslanja na koncepciju koju smo upoznali kod razvoja ostalih tipova aplikacija, vjerujemo da nećete imati problema sa samostalnim proučavanjem. Uvijek morate imati na umu da je .NET Compact Framework prilično osakaćena inačica .NET Frameworka te da nisu podržane mnoge stvari koje bismo mogli smatrati osnovnima. U takvim se okolnostima morate snalaziti koristeći ono što je dostupno i nadograđujući podržanu funkcionalnost. Takav način rada zahtijeva određeno programersko iskustvo i domišljatost, a strpljivijima ostaje i nada da će sljedeća inačica .NET Compact Frameworka donijeti novosti koje će riješiti dio problema.

Više informacija, zanimljivosti i specifičnosti o .NET Compact Frameworku možete pronaći na adresama http://msdn.microsoft.com/mobility, http://samples.gotdotnet.com/quickstart/CompactFramework/ i brojnim drugim web-stranicama te grupama na Usenetu.

DODATAK B

Otklanjanje grešaka

U ovom poglavlju:

- Otklanjanje grešaka u programima uz pomoć Visual Studija .NET
- Korištenje prekidnih točki
- Praćenje poziva metoda u prozoru *Call Stack*
- Izvršavanje naredbi preko prozora *Command*
- Praćenje vrijednosti varijabli

Kao što ste mogli vidjeti u druženju s .NET-om kroz cijelu knjigu, greške su vrlo stvarne i ne događaju se nekom drugom. Na svu sreću, Visual Studio .NET nudi vam izvrsne mehanizme njihova pronalaženja, identificiranja i znatno olakšava njihovo otklanjanje.

U nastavku ovog poglavlja upoznat ćete osnovne načine *debugiranja* aplikacija odnosno upoznat ćete proces otklanjanja *bugova* i grešaka koji se obavlja detaljnim praćenjem rada aplikacije, promjene varijabli i slično. Sve što će biti prikazano u ovom poglavlju vezano je uz Visual Studio .NET, što znači da prikazane metode možete koristiti pri razvoju svih tipova aplikacija, od prozorskih preko web-aplikacija sve do web-servisa.

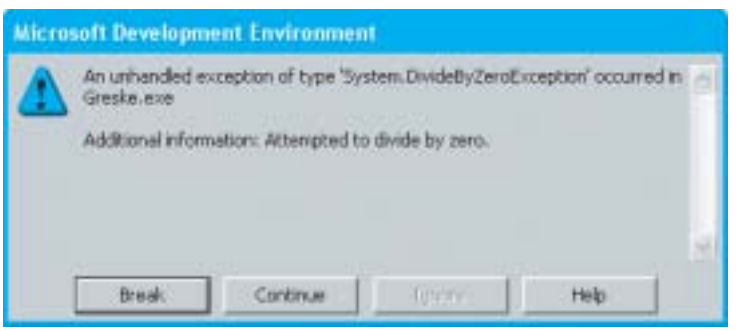
Praćenje rada aplikacije

Za upoznavanje s dijelovima sučelja koji pomažu pri pronalaženju i otklanjanju grešaka trebat će nam i program na kojem ćemo sve to testirati. Naravno, on će u sebi imati grešaka, no i dijelova kôda koji će poslužiti isključivo prikazivanju mogućnosti.

```
private void Form1_Load(object sender, System.EventArgs e)
{
    int a = 20;
    int b = 10;

    sqlDataAdapter1.Fill(dataSet11);

    if (b == 10)
    {
        b = b - 5;
    }

    double c = a / (b - 5);

    textBox1.Text = c.ToString();
}
```

Nemojte da vas zbune objekti *DataAdapter* i *DataSet* – oni su prethodno definirani putem Visual Studio .NET sučelja i služit će da pokažemo neke mogućnosti sučelja. Isto tako, prethodna metoda je definirana u prozorskoj aplikaciji, no sve što će biti prikazano može biti primijenjeno i na druge tipove aplikacija.

U prethodnom kodu postoji očita greška – pri računanju varijable *c* pokušavamo dijeliti s nulom (što bi rezultiralo s beskonačnim brojem), što pri izvođenju aplikacije rezultira greškom na slici B-1.



Slika B-1: ***Greška u aplikaciji kao posljedica pokušaja dijeljenja s nulom***

Da biste mogli *debugirati* aplikaciju, trebate je pokrenuti odabirom opcije *Debug – Start*. Ukoliko se aplikacija ispravno pokrene, možete se vratiti u Visual Studio – razvojna okolina će promijeniti svoje sučelje i pojavit će se niz novih prozora, koje ćemo u nastavku objasniti.

Ukoliko ne želite svoje aplikacije pri pokretanju *debugirati*, mnogo je efikasnije pokrenuti opciju *Start Without Debugging* (CTRL+F5) jer će se tad aplikacija brže pokrenuti.

Prekidne točke

No za početak, potrebno je ukratko objasniti prekidne točke (engl. *breakpoint*) koje ćemo koristiti za kontrolu izvršavanja kôda. One omogućavaju zaustavljanje izvršavanja aplikacija kad se dođe do određene naredbe u kodu. Najjednostavniji način postavljanja prekidne točke je da kliknete na sivi rub uz liniju gdje je želite ubaciti. Na tom mjestu će se pojaviti crvena točka koja označava umetnutu prekidnu točku, a cijela naredba na kojoj se dešava prekid bit će obojana crvenom bojom.

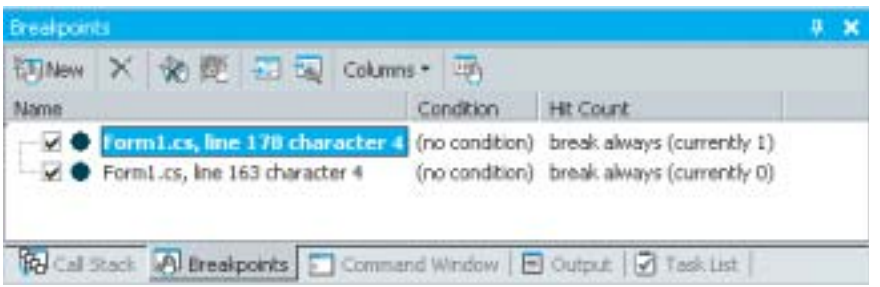
```
            if (b == 10)
            {
                b = b - 5;
            }

            double c = a / (b - 4);

            textBox1.Text = c.ToString();
        }
    }
}
```

Slika B-2:
Prekidna točka postavljena je na problematičnu naredbu.

Prekidna točka zaustavlja izvršavanje programa netom prije naredbe u kojoj je ona postavljena.

Pokrenete li aplikaciju u načinu *Debug*, ona će se zaustaviti u zadanom trenutku, a fokus će se vratiti u Visual Studio .NET. Popis svih prekidnih točki možete vidjeti u prozoru *Breakpoints* prikazanom na slici B-3. Ukoliko ne vidite navedeni prozor, uključite ga opcijom *Debug – Windows – Breakpoints*.

Slika B-3: ***Popis prekidnih točki u programu prikazan je u prozoru Breakpoints.***



U programu možete imati proizvoljan broj prekidnih točki odnosno možete namjestiti da se izvršavanje aplikacije zaustavi na više različitih mjesta u programu.

Izvršavanje kôda naredbu po naredbu

Prekidne točke postavljat ćete u problematičnim dijelovima aplikacije, tj. u onima za koje sumnjate da rezultiraju greškom, no ne možete je otkriti. Nakon što zaustavite izvođenje aplikacije, iskoristit ćete različite dijelove sučelja koji će biti objašnjeni u nastavku poglavlja.

No prekidne točke mogu služiti i za kontrolu izvršavanja kôda. Pokrenemo li aplikaciju s prethodnim kôdom i postavljenom prekidnom točkom na *if*-naredbi, Visual Studio .NET će zaustaviti izvođenje aplikacije i prikazati žutu strelicu koja ukazuje na trenutno aktualnu naredbu (i sama naredba će biti u žutoj boji).

Slika B-4: ***Izvršavanje aplikacije zaustavljeno je na prekidnoj točki – žuta strelica i boja označavaju trenutno aktualnu naredbu.***

```
if (b == 10)
{
    b = b - 5;
}

double c = a / (b - 4);
```

U trenutku kad je zaustavljena aplikacija, na raspolaganju vam stoje tri opcije za kontrolu izvršavanja kôda – *Step Over*, *Step Out* i *Step Into*. Sve se nalaze u izborniku *Debug*.

Preporučljivo je naučiti tipkovničke kratice spomenutih opcija – *Step Over* možete izvršiti pritiskom na F10, *Step Into* pritiskom na F11, a *Step Out* pritiskom na SHIFT+F11. Korištenjem tih kratica mnogo ćete lakše kontrolirati izvršavanje kôda nego stalnim odlaskom u izbornik *Debug* ili klikanjem po plutajućem izborniku *Debug*.

Step Into vam služi za ulazak jednu razinu dublje u kôdu, ili barem prelazak na sljedeću naredbu. Tu ćete opciju koristiti za izvršavanje svake naredbe u kodu. Njenim odabirom izvršit će se trenutno aktualna naredba, a žuta oznaka će se postaviti na sljedeću naredbu. Ukoliko je sljedeći na redu za izvršavanje poziv neke metode, pokretanjem *Step Into* opcije ući ćete u tu metodu i izvršavati kompletan njen kôd naredbu po naredbu.

Suprotna od opcije *Step Into* je opcija *Step Over*. Ukoliko je sljedeći na redu poziv neke metode, pokretanjem ove opcije preskačete *debugiranje* te metode i svih njenih naredbi. No pripazite – to ne znači da se ta metoda neće izvršiti, već jednostavno nećete dobiti priliku za izvršavanje svake njene pojedine naredbe. Ta metoda će se odjedanput izvršiti, a za aktualnu naredbu bit će postavljena prva sljedeća naredba poslije poziva metoda. To je idealna opcija za slučaj kad ste sigurni da neka metoda ispravno radi i ne želite trošiti vrijeme na lupanje F10 za svaku njenu naredbu.

Step Out služi, kao što i ime kaže, za izlazak iz trenutne metode. No kao i kod opcije *Step Over*, to ne znači da se ostatak kôda metode koja se *debugira* neće izvršiti. Baš suprotno, on će se kompletno izvršiti, no vi nećete trebati ručno izvršavati svaku pojedinu naredbu. Dakle, ova je opcija idealna za situacije kad ste izvršili sve problematične naredbe u metodi i ostale su samo neke za koje ste sigurni da rade, a želite završiti s njenim *debugiranjem*. Odabirom ove opcije aktualna naredba za izvršavanje postaje sljedeća naredba nakon poziva metode koju ste *debugirali*.

Praćenje poziva metoda

Da biste se lakše snalazili s opcijom *Step Out*, na raspolaganju vam stoji prozor *Call Stack*. Ukoliko ga ne vidite pri *debugiranju* aplikacije, kliknite na *Debug – Windows – Call Stack*. Kao što mu i samo ime kaže, radi se o popisu poziva metoda predstavljenih u obliku stoga.

Stog je struktura organizirana po LIFO (*Last In, First Out*) principu, što znači da će element koji je zadnji dodan doći na početak strukture, ali će zato morati biti prvi maknut s nje. Element koji je prvi ušao u strukturu bit će na dnu i zato će zadnji moći biti s nje izbrisan.

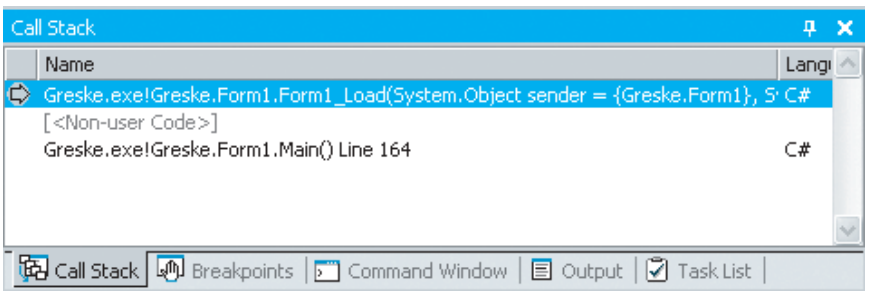
Primijenite li navedenu definiciju stoga i njegove LIFO strukture bit će vam jasnije kako je organiziran prozor s popisom poziva metoda. Dakle, zamislite situaciju u kojoj imate metodu iz koje pozivate neku drugu metodu, a iz nje pak pozivate neku treću metodu, primjerice:

```
public void PrvaMetoda() {
   DrugaMetoda();
}

public void DrugaMetoda() {
   TrecaMetoda();
}

public void TrecaMetoda() {
   // neki kôd
}
```

Metoda koja je prva pozvana (recimo da je to *PrvaMetoda*) bit će na dnu liste. Iz nje je pak pozvana *DrugaMetoda* te je ona stavljena na mjesto na listi iznad nje. Iz nje je pak pozvana treća metoda, koja će tako biti stavljena na vrh liste, iznad druge dvije.

```
TrecaMetoda
DrugaMetoda
PrvaMetoda
```

Nakon što se završi izvršavanje treće metode, ona se miče s vrha liste i na njoj ostaju samo dvije metode, *PrvaMetoda* i *DrugaMetoda*. Nakon što se izvrši druga metoda, i ona se miče s liste i ostaje samo *PrvaMetoda*. Sad razumijete strukturu LIFO – *TrecaMetoda* je zadnja došla, no prva je otišla, jer je njeno izvršavanje prvo završilo.



Slika B-5: ***Prikaz prozora Call Stack – metoda Main je prva pozvana, a zatim je pozvana metoda Form_Load***

Dakle, pri pozivanju opcije *StepOut* pogledom na prozor *Call Stack* uvijek možete znati u kojoj ćete metodi nastaviti s izvršavanjem svake pojedine naredbe – u onoj koja je navedena prva

ispod one u kojoj se trenutno nalazite. Kako će na vrhu prozora *Call Stack* biti navedena uvijek trenutna metoda, pozivanjem opcije *StepOut* nastavit ćete s izvršavanjem u drugoj metodi s liste, u naredbi koja dolazi odmah poslije poziva trenutne metode.

Prozor *Call Stack* može poslužiti i u složenim aplikacijama kako biste znali otkud je došao poziv trenutnoj metodi. Kliknete li na neku od metoda s popisa, u kodu će se pojaviti zelena oznaka koja će označavati trenutnu poziciju u toj metodi. Primjerice, ako istu metodu pozivate s više mjesta, klikom na metodu koja je ispod nje na listi u prozoru *Call Stack* možete doznati odakle je potekao njen poziv.

Izvršavanje naredbi za vrijeme izvršavanja aplikacije

Jedan od najkorisnijih prozora u procesu pronalaženja i otklanjanja grešaka je svakako prozor *Command*. Koliko ste puta u svom kodu htjeli izvršiti poziv neke metode samo da vidite da li ona vraća ispravnu vrijednost? Koliko ste puta u kodu htjeli promijeniti vrijednost neke varijable samo da vidite kako će dalje teći izvršavanje aplikacije? Ukoliko niste koristili prozor *Command*, bili ste osuđeni na mijenjanje kôda i dodavanje kôda za provjeru, kompajliranje cijele aplikacije i njeno ponovno pokretanje, što je bio ipak spor proces za jednostavna testiranja.

Korištenjem prozora *Command* možete izvršavati različite naredbe koje će direktno utjecati na izvršavanje kôda, odnosno rezultat će biti isti kao da su te naredbe izvedene u samom kodu aplikacije. Da biste prikazali prozor *Command*, ukoliko ga ne vidite kao jedan od ponuđenih prozora na dnu ekrana, odaberite njegovo prikazivanje iz izbornika *Debug – Windows*.

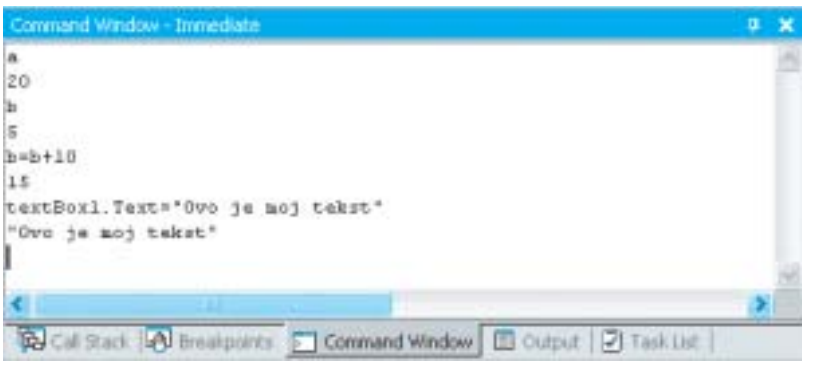
Da biste saznali vrijednost neke varijable u kodu, jednostavno upišite njeno ime u prozor *Command* i u sljedećem retku će se pojaviti njena vrijednost. Da biste promijenili njenu vrijednost, iskoristite znak jednakosti i upišite njenu novu vrijednost, koja čak može biti izračunata iz trenutnih vrijednosti drugih varijabli. U narednom prikazu kôda podebljano su prikazane naredbe koje su upisane u prozor *Command*, a normalnim stilom tekst koji je rezultat upisanih naredbi.

```
b
10
b=b+10
20
b=a+b
40
dataSet11.Employees.Rows.Count
9
```

Primijetite kako u prozoru *Command* možete ispisivati i propitkivati vrijednosti jednostavnih i složenih objekata, poput broja redaka u tablici *Employees* u objektu *dataSet11*. Imajte na umu da promjena vrijednosti svih varijabli direktno utječe na kôd – postavite li tako prekidnu točku oglednom

programu prije dijeljenja s nulom u varijabli *c* i u iskoristite prozor *Command* za mijenjanje vrijednosti varijable *b* (promijenite je da ne bude jednaka 5), to će direktno utjecati na njenu vrijednost u programu i dijeljenje s nulom se neće dogoditi.

Pri ispisivanju svojstava objekata u prozoru *Command*, kao i pri pisanju kôda, na raspolaganju vam stoji IntelliSense, koji će vam automatski ponuditi izbor svih dostupnih svojstava i metoda nekog objekta kad upišete točku iza njegova imena.



Slika B-6: ***Korištenje prozora Command može bitno olakšati testiranje rada aplikacije.***

U prozoru *Command* mogli ste i pozivati različite metode i pratiti njihove rezultate. Pozivi metoda pišu se na isti način kao i u programu, a čak ćete uz pomoć tehnologije IntelliSense dobiti popis i opis parametara metode, što će vam dodatno olakšati testiranje.

Praćenje vrijednosti varijabli

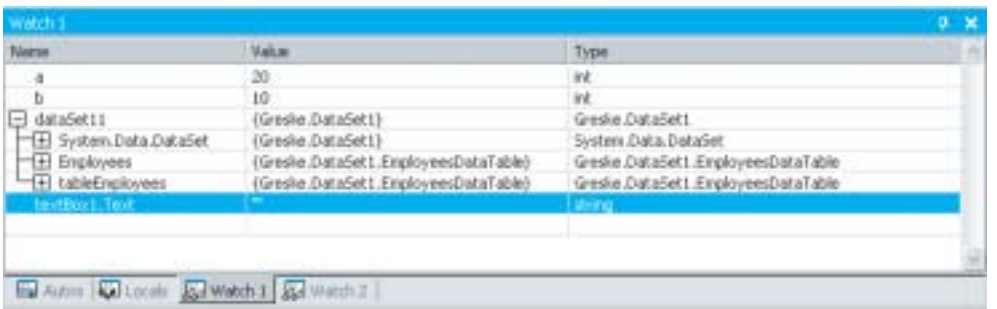
Želite li na jednostavan način pratiti vrijednosti varijabli korištenih u programu, a da ne morate ručno svaki put upisivati njihovo ime u prozoru *Command*, na raspolaganju vam stoji prozor *Watch*. Dapače, zbog preglednosti i mogućnosti praćenja više različitih skupova varijabli, možete prikazati čak četiri prozora *Watch* – uključite ih preko opcija dostupnih u izborniku *Debug – Windows – Watch* ili pritiskom tipkovničke kratice CTRL+ALT+W u kombinaciji s jednim od brojeva između 1 i 4.

Da biste pratili vrijednost neke varijable, samo upišite njeno ime u stupcu *Name* u prozoru *Watch*. No ne samo što možete pratiti vrijednosti jednostavnih varijabli, već i objekata – upišete li ime nekog objekta, dobit ćete popis svih njegovih svojstava koje potom možete pratiti kroz izvršavanje aplikacije, a ona će zbog preglednosti biti uvučena i dostupna na klik na znak "+" kraj imena objekta.

Preko prozora *Watch* također možete mijenjati vrijednosti svih prikazanih varijabli. Jednostavno kliknite u redak s varijablom u stupcu *Value* i upišite novu željenu vrijednost.

Korištenje prozora *Watch* idealno je za prvi korak u traženju greške – kad se uvjerite da su svi podaci ispravni i baš u obliku u kojem ih očekujete, svoju pažnju možete posvetiti programskoj logici.

Slika B-7:
Prozor Watch u akciji; uočite znak "+" pokraj pojedinih varijabli, što znači da one sadržavaju još čitav niz svojstava koja se mogu prikazati klikom na njega.



Prozor *Watch* će zapamtiti koje ste varijable upisali u njega i pri svakom sljedećem pokretanju aplikacije u *Debug* načinu rada one će biti prikazane u njemu, što znatno olakšava praćenje rada aplikacije. Stanja varijabli u prozoru *Watch* najbolje je pratiti korištenjem opisanih metoda za izvršavanje kôda naredbu po naredbu, jer tako imate potpunu kontrolu i nikakva vam promjena ne može umaći. Dapače, promjena vrijednosti varijabli koja se obavila u prethodnoj naredbi je u prozoru *Watch* označena crvenom bojom.

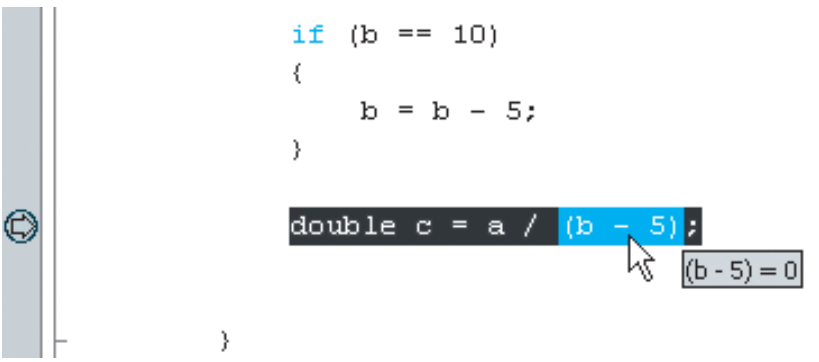
Uz prozor *Watch* u istoj skupini je i prozor *Autos* koji također služi za prikaz stanja varijabli, no u njemu se automatski prikazuju samo varijable koje su povezane s trenutno aktualnom linijom. Ukoliko niste prezahtjevni, pogledom na ovaj prozor uvijek možete vidjeti stanja najvažnijih varijabli koje se koriste pri izvođenju neke naredbe, što bi vam u većini slučajeva moglo biti dosta za pregled rada aplikacije.

Quick Watch

Osim praćenja stanja varijabli preko posebnog prozora *Watch*, na raspolaganju vam stoji i još jedna vrlo korisna funkcionalnost Visual Studija .NET nazvana *Quick Watch*. Naime, u procesu *debugiranja* možete na vrlo jednostavan način saznati vrijednost varijabli ili nekog izraza.

Trebate samo privući miša varijabli u kodu i iznad nje će se pojaviti njena vrijednost. Tako se možete mišem kretati po cijelom kodu aplikacije i propitkivati varijable o njihovoj vrijednosti.

No to nije sve – kao što smo rekli, možete pratiti i vrijednost nekog izraza, kao što je prikazano na slici B-8. Trebate samo označiti željeni izraz, a njegova vrijednost će vam se također pojaviti u oblačiću.

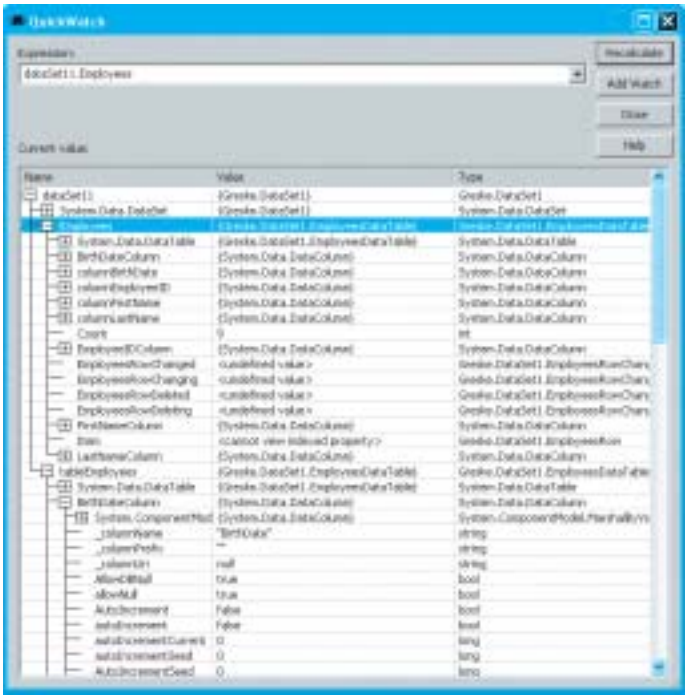


Slika B-8: ***Označavanjem nekog izraza Visual Studio će prikazati njegovu vrijednost.***

Kako se u oblačiću mogu prikazati samo jednostavne vrijednosti varijabli, što je prilično ograničenje ukoliko želite ispitivati svojstva objekata i svih njihovih vrijednosti, možete iskoristiti opciju *Quick Watch*. Kliknete li tako na neki objekt čiju vrijednost ne možete prikazati u jednostavnom oblačiću (primjerice, objekt *DataSet*) desnim gumbom miša, u izborniku možete odabrati opciju *Quick Watch*.

Praćenje rada aplikacije ispisivanjem poruka

Najjednostavniji je način praćenja rada aplikacije i – jer u nekom od svojih oblika postoji od prvih metoda programiranja – najpopularniji, ispisivanje različitih poruka u pojedinim dijelovima aplikacije. Najčešće su to jednostavne poruke koje trebaju svojim pojavljivanjem samo dati do znanja da se određeni dio kôda izvršio pa se ispisuju poruke poput “tu sam”, “radi” i slično. Naravno, često se i u takvom ispisivanju poruka ispisuju i vrijednosti nekih varijabli, no to ipak više ne morate raditi jer vam na raspolaganju stoje prozor *Watch* i opcija *Quick Watch*.



Slika B-9:
Detaljan prikaz nekog objekta tipa DataSet u prozoru Quick Watch

No zahvaljujući Visual Studiju više niste osuđeni na ispisivane poruka koje se pojavljuju u samoj aplikaciji (bilo kao neki tekst na web-stranici u web-aplikacijama ili kao poruka prikazana metodom *MessageBox.Show* u prozorskim aplikacijama). U Visual Studiju možete različite poruke ispisivati u prozoru *Output,* i to korištenjem *Debug.WriteLine* naredbe koja radi u svim tipovima aplikacija, jer nije vezana uz samo sučelje aplikacije, već uz razvojnu okolinu.

Da biste mogli koristiti klasu *Debug* i njenu metodu *WriteLine* (kao i druge metode iz klase), u svoju aplikaciju trebate uključiti *namespace System.Diagnostics*.

Evo kako možete koristiti klasu *Debug* za ispisivanje poruka:

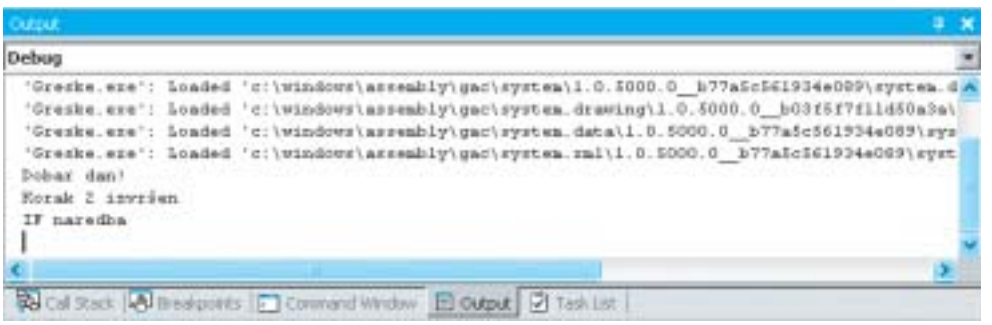
```
Debug.WriteLine("Korak 2 izvršen!");
```

Često se ispisivanje tih poruka koristi i u *if*-naredbama jer se želi ispisati posebna poruka ako se ušlo u *if*-blok, odnosno ako se ušlo u *else*-blok (tj. želi se saznati da li je uvjet u *if*-naredbi ispunjen).

Klasa *Debug* nudi vam jednostavniji način – možete iskoristiti metodu *WriteLineIf* u kojoj kao prvi parametar navodite uvjet koji treba provjeriti. Ukoliko je taj uvjet istinit, poruka će se ispisati, a u protivnom, naravno, neće.

```
Debug.WriteLineIf(a > b, "A je veći od B!");
Debug.WriteLineIf(textBox1.Text != "", "Nešto piše u TextBoxu!");
```

Slika B-10:
Poruke ispisane metodama klase Debug pojavit će se u prozoru Output.



U prozoru *Output* ispisuju se i poruke vezane uz izvršavanje programa (primjerice, o učitanim *assemblyjima*).

Korištenjem ovdje opisanih opcija možete značajno olakšati razvoj aplikacija. Greške su nešto normalno i očekivane su pri razvoju i najjednostavnijih aplikacija. Stoga ne treba stavljati naglasak na izradu savršenog kôda bez grešaka (zar to uopće postoji?), već na pronalaženje problematičnih dijelova kôda i traženju grešaka. Zahvaljujući ovdje opisanim metodama to više nije bauk i predstavlja izrazito jednostavan i moćan proces koji možete koristiti u svakoj fazi izrade aplikacija. Što se bolje upoznate s mogućnostima razvojne okoline Visual Studija, izrada aplikacija bit će vam lakša i ugodnija.

DODATAK C

Pomoćni alati za .NET

U ovom poglavlju:

- Čemu služe pomoćni alati za .NET
- Popis svih dostupnih pomoćnih alata

Kako bi izrada, implementacija i upravljanje aplikacijama i komponentama u .NET Frameworku bili što jednostavniji, Microsoft je razvio set alata pod nazivom .NET Framework SDK Tools. Svi se (osim dva kod kojih ćemo to naglasiti) koriste iz komandne linije, no kako su smješteni u mapi "c:\Windows\Microsoft.NET\SDK-\v1.1\Bin" vjerojatno neće biti dostupni samim upisom imena u komandnu liniju.

Tri su rješenja ovoga problema: prvi i najjednostavniji je koristiti prečicu koju je Visual Studio .NET kreirao tijekom instalacije u svojoj programskoj grupi (podgrupa Tools), koja će automatski podesiti potrebne parametre za korištenje tih alata. Drugi način je podesiti te parametre ručno ili pak (treća varijanta) pokrenuti datoteku "vcvars32.bat" koja će to učiniti za vas, a nalazi se u već spomenutoj mapi.

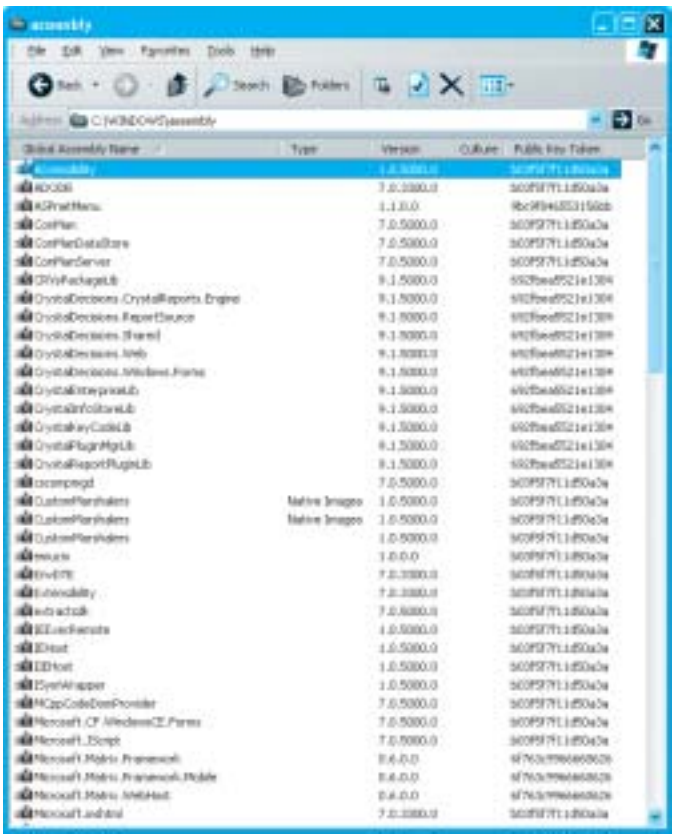
Popis alata

Mi ćemo u nastavku pobrojati sve alate koji su dostupni u .NET Framework SDK 1.1 te ukratko objasniti njihovu namjenu. Ukoliko vam neki od njih zatreba u radu, detaljnije informacije možete pronaći u dokumentaciji, na putanji .NET Development > .NET Framework SDK > .NET Framework > Tools and Debugger > .NET Framework Tools.

Radi lakšeg snalaženja, alati su podijeljeni u četiri skupine: alati za konfiguraciju i *deployment*, alati za *debugiranje*, sigurnosni alati i ostali.

Alati za konfiguraciju i *deployment*

- **ASP.NET IIS Registration Tool** (Aspnet_regiis.exe) – omogućava administratoru ili instalacijskom programu registriranje ASP.NET-a na serveru kao i neke druge konfiguracijske radnje.



Slika C-1:
Assembly Cache Viewer

- **Assembly Cache Viewer** (Shfusion.dll) – omogućava pregled i upravljanje sadržajem Global Assembly Cachea kroz Windows Explorer (taj se alat ne može koristiti iz komandne linije, no dovoljno je u Windows Exploreru doći u mapu gdje je smješten GAC i on će biti pokrenut).

Assembly Linker (Al.exe) – generira datoteku s manifestom *assemblyja* iz jedne ili više datoteka s resursima ili kôdom u MSIL-u

Assembly Registration Tool (Regasm.exe) – na temelju metapodataka u *assemblyju* ubacuje potrebne zapise u *registry,* što omogućava starijim aplikacijama da koriste klase .NET Framework kao da su COM-ovi

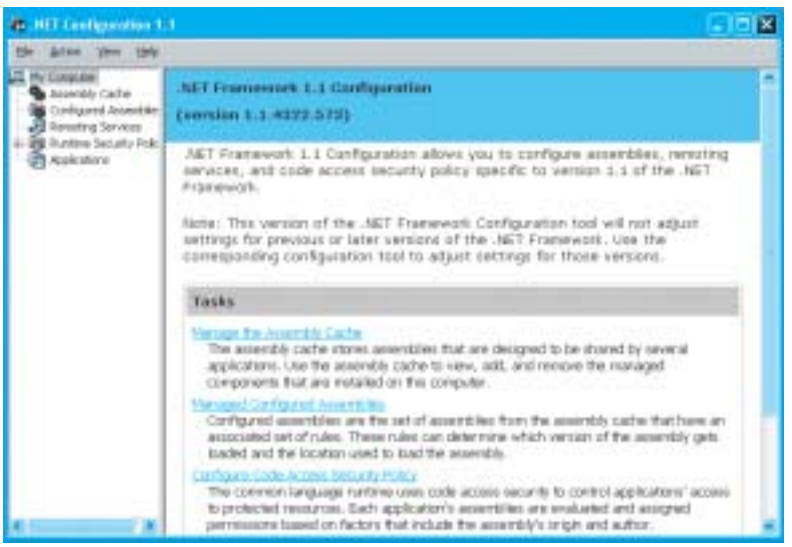
Assembly Binding Log Viewer (Fuslogvw.exe) – pomaže pri dijagnosticiranju zašto .NET Framework ne može pronaći *assembly* za vrijeme izvršavanja

Global Assembly Cache Tool (Gacutil.exe) – omogućava pregled i upravljanje sadržajem Global Assembly Cachea; slično kao Assembly Cache Viewer, samo što se koristi iz komandne linije

Installer Tool (Installutil.exe) – omogućava instalaciju i deinstalaciju serverskih resursa

Isolated Storage Tool (Storeadm.exe) – ispisuje ili briše sva postojeća spremišta trenutno prijavljenog korisnika

Native Image Generator (Ngen.exe) – izrađuje *native image* iz *assemblyja* i instalira ga u pripadajući *cache*



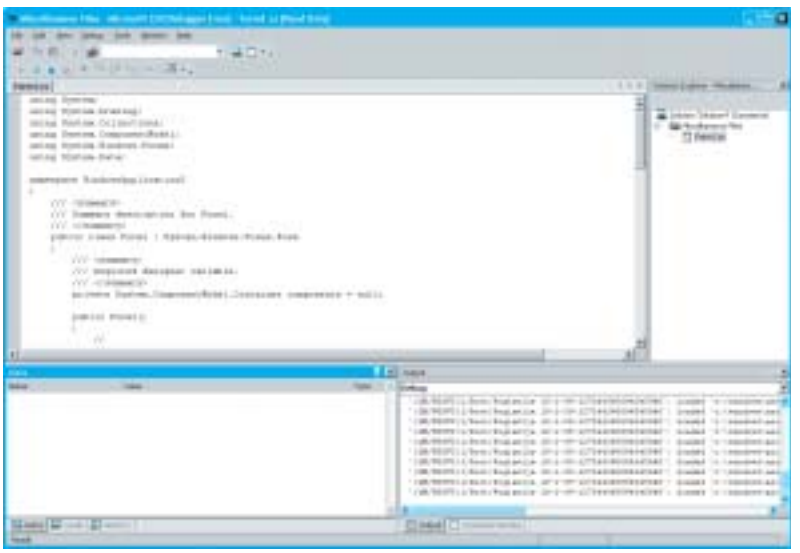
Slika C-2: *.NET Framework Configuration Tool*

.NET Framework Configuration Tool (Mscorcfg.msc) – omogućava upravljanje sigurnosnom politikom i aplikacijama kroz grafičko sučelje

- **.NET Services Installation Tool** (Regsvcs.exe) – služi za povezivanje COM+ 1.0 aplikacija s upravljanim klasama
- **Soapsuds Tool** (Soapsuds.exe) – pomaže pri kompajliranju klijentskih aplikacija koje komuniciraju s web-servisima koristeći tehniku *remoting*
- **Type Library Exporter** (Tlbexp.exe) – stvara COM *type library* iz CLR *assemblyja*
- **Type Library Importer** (Tlbimp.exe) – pretvara definicije iz COM *type libraryja* u upravljani format metapodataka
- **Web Services Description Language Tool** (Wsdl.exe) – stvara kôd za web-servise iz WDSL-a, XSD-a i datoteka s nastavkom .discomap (detaljnije objašnjeno u devetom poglavlju)
- **Web Services Discovery Tool** (Disco.exe) – otkriva adrese web-servisa smještenih na web-poslužitelju
- **XML Schema Definition Tool** (Xsd.exe) – kreira XML-*scheme* prema jeziku XSD

Alati za *debugiranje*

- **Microsoft CLR Debugger** (DbgCLR.exe) – *debugger* s grafičkim sučeljem (nije dostupan iz komandne linije, nalazi se u mapi "C:\Windows\Microsoft.NET\FrameworkSDK-\GuiDebug" ili C:\Program Files\Microsoft Visual Studio .NET 2003\SDK\v1.1\GuiDebug)



Slika C-3: ***Microsoft CLR Debugger***

- **Runtime Debugger** (Cordbg.exe) – komandno-linijski *debugger*

Sigurnosni alati

- **Certificate Creation Tool** (Makecert.exe) – generira certifikate X.509 za testiranje
- **Certificate Manager Tool** (Certmgr.exe) – upravlja certifikatima, CTL-ovima i CRL-ovima
- **Certificate Verification Tool** (Chktrust.exe) – provjerava valjanost datoteke potpisane certifikatom X.509
- **Code Access Security Policy Tool** (Caspol.exe) – omogućava pregled i promjenu sigurnosne politike za računalo, korisnika i *enterprise*
- **File Signing Tool** (Signcode.exe) – potpisuje izvršnu datoteku (*portable executable* – PE) digitalnim potpisom Authenticode
- **Permissions View Tool** (Permview.exe) – prikazuje minimalne, optimalne i odbijene setove dozvola koje navedeni *assembly* zahtijeva
- **PEVerify Tool** (PEverify.exe) – provjerava *type safety* i valjanost metapodataka određenog *assemblyja*
- **Policy Migration Tool** (Migpol.exe) – prebacuje sigurnosnu politiku između dvije kompatibilne verzije .NET Frameworka
- **Secutil Tool** (Secutil.exe) – vadi jako ime ili certifikat izdavača Authenticode iz *assemblyja*
- **Set Registry Tool** (Setreg.exe) – omogućava promjenu postavki u *registryju* vezanih uz provjeru certifikata
- **Software Publisher Certificate Test Tool** (Cert2spc.exe) – stvara Software Publisher's Certificate (SPC) iz jednog ili više certifikata X.509 (samo za testiranje)
- **Strong Name Tool** (Sn.exe) – pomaže pri stvaranju jako imenovanih *assemblyja*

Ostali alati

- **Common Language Runtime Minidump Tool** (Mscordmp.exe) – stvara datoteku koja sadrži informacije potrebne za analizu sustava
- **License Compiler** (Lc.exe) – čita tekstualne datoteke koje sadrže podatke o licencama i stvara datoteku kompatibilnu sa CLR-om

- **Management Strongly Typed Class Generator** (Mgmtclassgen.exe) – omogućava brzo stvaranje *early-bound* klase za određenu Windows Management Instrumentation (WMI) klasu

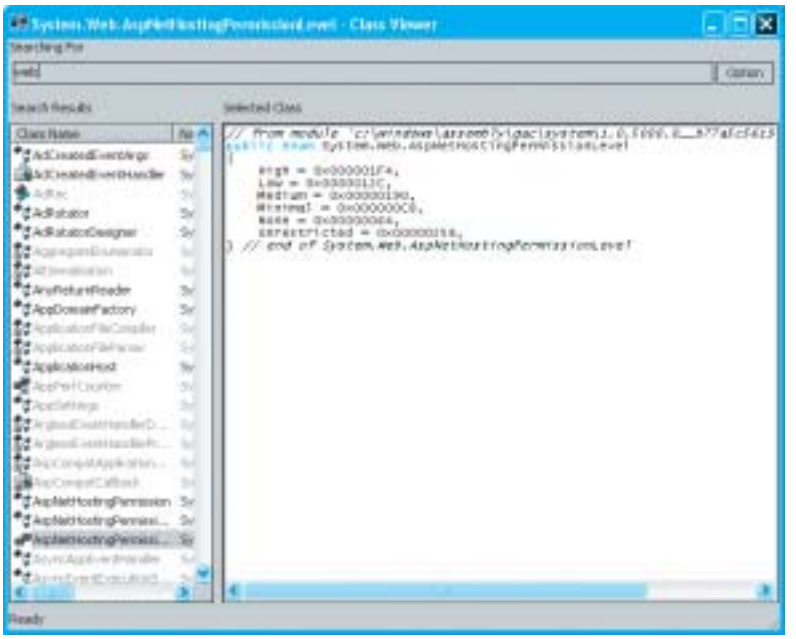
- **MSIL Assembler** (Ilasm.exe) – stvara izvršnu datoteku iz kôda pisanog u MSIL-u

- **MSIL Disassembler** (Ildasm.exe) – iz izvršne datoteke koja sadrži MSIL izvlači kôd u tekstualnu datoteku

- **Resource File Generator Tool** (Resgen.exe) – pretvara tekstualne i .resx datoteke u .NET-ove binarne datoteke s nastavkom .resources koje mogu biti uključene u izvršnu datoteku ili kompajlirane u *assemblyje*

- **Windows Forms ActiveX Control Importer** (Aximp.exe) – pretvara *type* definicije u COM *type libraryju* u kontrolu Windows Formsa

- **Windows Forms Class Viewer** (Wincv.exe) – pretražuje upravljane klase i prikazuje informacije o njima



Slika C-4: ***Windows Forms Class Viewer***

- **Windows Forms Resource Editor** (Winres.exe) – omogućava brzu i jednostavnu lokalizaciju prozorskih aplikacija

DODATAK D

DVD

U ovom poglavlju:

- Što se nalazi na pratećem DVD-u
- Što je to "probna verzija"

Uz ovu knjigu dobili ste i DVD na kojem se nalazi probna verzija Visual Studija .NET 2003 Professional. Kao što već sigurno znate, radi se o softveru za razvoj aplikacija na .NET tehnologiji koji je korišten u skoro svakom poglavlju ove knjige.

Slika D-1:
Instalacija paketa Visual Studio .NET 2003 Professional

O svim mogućnostima razvojnog alata Visual Studio .NET možete saznati na službenim web--stranicama na adresi http://msdn.microsoft.com/vstudio/. Tamo možete naći i niz članaka i drugih resursa koji vam mogu pomoći u programiranju i proširivanju znanja.

Korištenje probne verzije

Verzija Visual Studija .NET na DVD-u označena je kao probna verzija koju možete isprobavati 60 dana nakon instalacije. U tom periodu ćete moći koristiti sve mogućnosti Visual Studija i upotrijebiti ga za isprobavanje svih primjera iz ove knjige.

Probna verzija Visual Studija .NET 2003 Professional koja dolazi na pratećem DVD-u u potpunosti je funkcionalna. Nažalost, vrijeme besplatnog korištenja alata ograničeno je na spomenutih 60 dana.

Kad aktivirate Visual Studio nakon instalacije, za što vam je potrebna aktivna veza na Internet, dobit ćete licencu koja će vam omogućiti besplatno korištenje alata tijekom 60 dana. Naiđete li na neki problem u korištenju ili aktivaciji aplikacije, za informacije pogledajte na web-stranicu:

```
http://msdn.microsoft.com/vstudio/support/default.aspx
```

Što nakon 60 dana?

Nakon što prođe 60 dana za kojih možete bez ograničenja koristiti sve mogućnosti razvojne okoline, više je nećete moći koristiti. Naravno, poželite li je deinstalirati, to ćete moći učiniti bez problema.

Nakon što deinstalirate Visual Studio .NET po završetku 60-dnevnog probnog perioda, njegova ponovna instalacija neće imati učinka – probni period od 60 dana je već iskorišten i više nećete moći koristiti probnu verziju Visual Studija s priloženog DVD-a.

Želite li se ozbiljno baviti programiranjem, zasigurno ćete se odlučiti na kupnju pune verzije Visual Studija .NET. Oko odluke za kupnju pomoći će vam web-stranice Microsofta Hrvatska načinjene baš za tu svrhu – na njima možete naći informacije o verzijama Visual Studija .NET i o partnerima kod kojih ga možete kupiti.

http://www.microsoft.com/croatia/developers/

Rječnik

.ascx Nastavak datoteke u kojoj se nalazi ASP.NET-ova serverska kontrola.

.asmx Nastavak datoteke u kojoj se nalazi web-servis.

.aspx Nastavak datoteke u kojoj se nalazi ASP.NET skripta.

.NET Compact Framework Pojednostavljena inačica .NET Frameworka koja se koristi na malim uređajima.

apstraktne klase Kombinacija klasa i sučelja u kojima je dio funkcionalnosti implementiran dok će drugi dio odraditi onaj koji ju naslijedi.

ASP Microsoftova tehnologija za serversko skriptiranje, preteča ASP.NET-a.

assembly Aplikacija ili komponenta koja radi na .NET platformi.

autentikacija korisnika Proces utvrđivanja identiteta korisnika.

autorizacija korisnika Proces utvrđivanja ima li korisnik pravo pristupa nekom resursu.

Base Class Library (BCL) Skupina klasa koja sadrži osnovne funkcionalnosti koje koristimo u programiranju.

C# Novi programski jezik koji je Microsoft razvio u sklopu .NET platforme. Zamišljen je kao idealan spoj C++-a i Visual Basica, od kojih nasljeđuje njihove najbolje karakteristike.

cacheiranje Metoda spremanja generiranih stranica u memoriju kako se prilikom sljedećih zahtjeva ne bi trebale ponovo generirati.

code-behind v. pozadinski kôd

Common Language Runtime Dio .NET Frameworka koji se brine o kompajliranju i izvršavanju aplikacija.

cookie Podaci koje server sprema na računalu posjetitelja. Kad se posjetitelj vrati na iste web-stranice, njegov preglednik šalje kopiju *cookiea* natrag serveru. Koriste se za identifikaciju korisnika, spremanje personaliziranih podataka, itd.

delegati Poseban tip podataka koji u sebi sadrži poziv na neku metodu.

destruktor Metoda koja se poziva prilikom uklanjanja objekta iz memorije.

dijaloški okviri Prozori, forme koje sadrže određenu često korištenu funkcionalnost.

DNS *Domain Name System* je hijerarhijski sustav prema kojem svi serveri na Internetu imaju naziv domene (npr. bug.hr) i IP adresu (npr. 213.202.123.120). DNS sustav prevodi adrese domena, koje se koriste radi lakšeg pamćenja, u brojevne IP adrese koje služe za komunikaciju između računala.

Document Object Model (DOM) Programski model koji određuje kako se pristupa dokumentu i kako se on koristi.

događaji Trenuci u kojima se mijenjaju određene karakteristike kontrola. Ako na događaj vežemo metodu, onda će ona biti izvršena u trenutku nastupa tog događaja.

dokumentacija Nešto bez čega je nemoguće biti dobar programer – sadrži detaljne opise i specifikacije svega što nam prilikom programiranja može zatrebati. Vidi MSDN Library.

emulator Program koji simulira neki uređaj te tako omogućava testiranje aplikacija pisanih za taj uređaj.

funkcija v. metoda

Garbage collection Sustav koji se brine da nekorišteni objekti ne zauzimaju dragocjen pa ih briše i oslobađa memoriju.

Global Assembly Cache Mjesto gdje se spremaju komponente (*assemblyji*) koje trebaju biti dostupne svim aplikacijama na računalu. U njemu se nalaze sve bazne klase .NET Frameworka.

HTML *Hypertext Markup Language*. Jezik za opisivanje izgleda web-stranica.

HTTP *Hypertext Transfer Protocol*. Set pravila po kojem se Internetom prenose web-stranice i web-servisi.

IIS Microsoftov web server. Izvršava ASP i ASP.NET skripte, omogućava FTP pristup i pruža jednostavni SMTP poslužitelj za slanje poruka iz skripti.

IntelliSense Tehnologija u Visual Studiju koja pomaže prilikom pisanja kôda dovršavajući započete izraze, ukazujući na propuste i slično.

Intermediate Language (IL) vidi MSIL.

IP-adresa *Internet Protocol* adresa je 32-bitni (4 bajta) broj koji jedinstveno predstavlja neko računalo spojeno na Internet, a služi za komunikaciju između računala. Više informacija u 1. poglavlju.

iznimke Greška koja se pojavljuje prilikom nedozvoljenog korištenja vrijednosti.

klase Nacrti za kreiranje objekata.

kolačići Male tekstualne datoteke na računalu posjetitelja u kojima web-poslužitelj zapisuje podatke o posjetitelju.

komandna linija Tekstualno sučelje u kojem računalu dajemo jednu po jednu naredbu upisivanjem imena naredbe i navođenjem određenih parametara.

kompajliranje Proces u kojem se iz kôda pisanog u programskom jeziku generira kôd za izvršavanje na računalu.

konfiguracijske datoteke Datoteke, najčešće XML ili tekstualne u kojima definiramo postavke aplikacije.

konkatenacija Slijedno spajanje znakovnih nizova. Primjerice, rezultat konkatenacije nizova “dobar”, “ “ i “dan” je “dobar dan”.

konstruktor Metoda koja se izvršava prilikom stvaranja objekta iz neke klase.

kontrole Klase koje sadrže funkcionalnosti koje se često koriste. Zahvaljujući njima nije potrebno svaku funkcionalnost programirati *od nule*.

link (hyperlink) Osnovni element web-stranica. Predstavlja vezu između dvije stranice i povezuje cijeli web u jednu cjelinu.

localhost Ime koje predstavlja trenutno računalo. IP adresa lokalnog računala je 127.0.0.1, a njemu upućeni podaci ne putuju preko Interneta.

Longhorn Slijedeća klijentska inačica operativnog sustava Windows koja će se bazirati na .NET Frameworku.

metoda Komad kôda koji odrađuje neku funkcionalnost. Svaka metoda ima svoje ime, a, prema potrebi, može sadržavati jedan ili više ulaznih i izlaznih parametara.

Microsoft Intermediate Language (MSIL) Međujezik u koji se naš kôd kompajlira kako bi ga CLR mogao izvršiti.

MSDN Library Službena Microsoftova dokumentacija za .NET Framework i još hrpu drugih stvari.

namespace Putanja pomoću koje možemo referencirati članove neke klase.

nasljeđivanje Čin kreiranja nove klase tako da se preuzmu karakteristike postojeće.

neupravljani kôd Kôd koji se ne izvršava unutar .NET Frameworka.

newsgrupe Forumi na Internetu namijenjeni diskusijama o određenoj temi. *Newsgrupa* se sastoji od niza članaka (*postova*) i odgovora. Svaka *newsgrupa* ima svoj naziv iz kojeg je vidljiva tematika same grupe.

Northwind Ogledna baza podataka koju možete naći u svim Microsoftovim alatima za baze, a koja se često koristi za primjere.

ODBC *Open Database Connectivity* je sučelje koje omogućava aplikacijama zasnovanima na Windowsima pristup bazama podataka.

piksel Osnovna grafička jedinica (točka) za prikaz nekog sadržaja ili slike na ekranu.

poslužitelj Računalo koje odgovara na zahtjeve klijenta. Primjerice, web server prima zahtjeve za web-stranicama i šalje ih klijentu.

pozadinski kôd Programski kôd koji se nalazi u zasebnoj datoteci da se ne bi miješao s opisom sučelja.

prekoračenje Čin kojim nakon nasljeđivanja klase radimo vlastitu funkcionalnost za nekog već postojećeg člana i na taj ga način *prekoračujemo*.

preopterećenje Stvaranje više članova iste klase koji imaju različite funkcionalnosti, no pod istim imenom.

projekt Termin koji se koristi u Visual Studiju za kreiranje *assemblyja*.

RGB *Red-Green-Blue* je model kojim se opisuju boje na računalnom zaslonu. Svaka se boja može predstaviti kao zbroj tri različite boje – crvene, zelene i plave – a njihov udio u konačnoj boji mjeri se brojem između 0 i 255.

server v. poslužitelj

sintaksa Skup pravila koji propisuje kako se pišu naredbe u nekom programskom jeziku.

SOAP *Simple Object Access Protocol*. Protokol kojim komuniciraju web-servisi.

Solution Explorer Pomoćni prozor koji nudi popis svih datoteka i referenci koje koristimo u projektu.

spremljena procedura Procedura spremljena na SQL Serveru koja vrši neku radnju nad bazom podataka. Brža je od običnih SQL upita, njom možemo upravljati preko ulaznih, a vrijednosti dobivati preko izlaznih parametara.

SQL Server Najjača Microsoftova baza podataka koja se koristi u brojnim okolnostima.

SQL *Structured Query Language* je standardni jezik koji se koristi za rad s relacijskim bazama podataka. Služi za dohvaćanje zapisa iz baze, njihovo mijenjanje, dodavanje novih, brisanje postojećih, itd.

svojstva Parametri kojima određujemo karakteristike kontrola i objekata.

tag Tzv. "naredba" u HTML kôdu koja služi za opisivanje nekog sadržaja.

TaskList Pomoćni prozor Visual Studija koji može služiti kao podsjetnik na stvari koje još treba učiniti.

TCP/IP *Transmission Control Protocol / Internet Protocol* je niz standardnih protokola za razmjenu podataka preko mreže. Predstavlja temeljni protokol za komunikaciju računala na Internetu.

Toolbox Pomoćni prozor Visual Studija u kojem se nalaze sve dostupne kontrole.

učahurivanje Koncept koji govori da je implementacija objekta (klase) u potpunosti neovisna o njegovom sučelju.

upravljani kôd Kôd koji se u potpunosti izvršava unutar .NET Frameworka.

Usenet v. newsgrupe

validacija Proces analiziranja podataka i utvrđivanja slažu li se s prethodno postavljenim zahtjevima.

varijable "Mjesta" u memoriji gdje spremamo određene vrijednosti kako bismo ih mogli kasnije u kôdu koristiti.

Visual Studio .NET Razvojna okolina za pisanje aplikacija na .NET Frameworku.

višeoblič̨je Mogućnost da objekti (klase) na različite načine implementiraju ista sučelja.

W3C *World Wide Web Consortium.* Organizacija koje se brine o razvoju i standardizaciji brojnih protokola vezanih uz Internet.

web-aplikacija Skup svih web-skripti na istom *siteu* ili u istoj mapi koje osim adrese dijele postavke, sesije, kolačiće i slično.

web-servis Standardiziran način komunikacije među aplikacijama preko Interneta.

Whidbey Kôdno ime sljedeće inačice Visual Studija .NET koja će biti uz bok .NET Frameworku 2.0.

WSDL *Web Services Description Language*. Jezik temeljen na XML-u namijenjen opisivanju karakteristika web-servisa.

XML *Extensible Markup Language*. Izuzetno fleksibilan jezik za opisivanje podataka koji se koristi u brojnim tehnologijama, uključujući i web-servise, prilikom pristupa bazama podataka i slično.

XPath Jezik kojim opisujemo kako locirati i procesirati elemente u XML-dokumentima.

XSLT *Extensible Stylesheet Language Transformations*. Tehnologija pomoću koje transformiramo strukturu XML-dokumenta u drugi XML-dokument (treba li reći – promijenjene strukture).

Kazalo

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

ZABILJEŠKE

ZABILJEŠKE

ZABILJEŠKE

ZABILJEŠKE

BUG d.o.o.

za novinsko-nakladničku djelatnost
Adresa: Tratinska 36/I, 10000 Zagreb
Matični broj: 3926443
Telefon: 01/3821 555
Fax: 01/3821 669
Direktor: Aron Paulić

SysPrint d.o.o.

za nakladničku djelatnost
Adresa: XIV. trokut 8A, 10020 Zagreb
Matični broj: 3602486
Telefon: 01/6558 740
Fax: 01/6558 741
Direktor: Robert Šipek

Zagreb, 2004.

Rječnik

.ascx Nastavak datoteke u kojoj se nalazi ASP.NET-ova serverska kontrola.

.asmx Nastavak datoteke u kojoj se nalazi web-servis.

.aspx Nastavak datoteke u kojoj se nalazi ASP.NET skripta.

.NET Compact Framework Pojednostavljena inačica .NET Frameworka koja se koristi na malim uređajima.

apstraktne klase Kombinacija klasa i sučelja u kojima je dio funkcionalnosti implementiran dok će drugi dio odraditi onaj koji ju naslijedi.

ASP Microsoftova tehnologija za serversko skriptiranje, preteča ASP.NET-a.

assembly Aplikacija ili komponenta koja radi na .NET platformi.

autentikacija korisnika Proces utvrđivanja identiteta korisnika.

autorizacija korisnika Proces utvrđivanja ima li korisnik pravo pristupa nekom resursu.

Base Class Library (BCL) Skupina klasa koja sadrži osnovne funkcionalnosti koje koristimo u programiranju.

C# Novi programski jezik koji je Microsoft razvio u sklopu .NET platforme. Zamišljen je kao idealan spoj C++-a i Visual Basica, od kojih nasljeđuje njihove najbolje karakteristike.

cacheiranje Metoda spremanja generiranih stranica u memoriju kako se prilikom sljedećih zahtjeva ne bi trebale ponovo generirati.

code-behind v. pozadinski kôd

Common Language Runtime Dio .NET Frameworka koji se brine o kompajliranju i izvršavanju aplikacija.

cookie Podaci koje server sprema na računalu posjetitelja. Kad se posjetitelj vrati na iste web-stranice, njegov preglednik šalje kopiju *cookiea* natrag serveru. Koriste se za identifikaciju korisnika, spremanje personaliziranih podataka, itd.

delegati Poseban tip podataka koji u sebi sadrži poziv na neku metodu.

destruktor Metoda koja se poziva prilikom uklanjanja objekta iz memorije.

dijaloški okviri Prozori, forme koje sadrže određenu često korištenu funkcionalnost.

DNS *Domain Name System* je hijerarhijski sustav prema kojem svi serveri na Internetu imaju naziv domene (npr. bug.hr) i IP adresu (npr. 213.202.123.120). DNS sustav prevodi adrese domena, koje se koriste radi lakšeg pamćenja, u brojevne IP adrese koje služe za komunikaciju između računala.

Document Object Model (DOM) Programski model koji određuje kako se pristupa dokumentu i kako se on koristi.

događaji Trenuci u kojima se mijenjaju određene karakteristike kontrola. Ako na događaj vežemo metodu, onda će ona biti izvršena u trenutku nastupa tog događaja.

dokumentacija Nešto bez čega je nemoguće biti dobar programer – sadrži detaljne opise i specifikacije svega što nam prilikom programiranja može zatrebati. Vidi MSDN Library.

emulator Program koji simulira neki uređaj te tako omogućava testiranje aplikacija pisanih za taj uređaj.

funkcija v. metoda

Garbage collection Sustav koji se brine da nekorišteni objekti ne zauzimaju dragocjen pa ih briše i oslobađa memoriju.

Global Assembly Cache Mjesto gdje se spremaju komponente (*assemblyji*) koje trebaju biti dostupne svim aplikacijama na računalu. U njemu se nalaze sve bazne klase .NET Frameworka.

HTML *Hypertext Markup Language*. Jezik za opisivanje izgleda web-stranica.

HTTP *Hypertext Transfer Protocol*. Set pravila po kojem se Internetom prenose web-stranice i web-servisi.

IIS Microsoftov web server. Izvršava ASP i ASP.NET skripte, omogućava FTP pristup i pruža jednostavni SMTP poslužitelj za slanje poruka iz skripti.

IntelliSense Tehnologija u Visual Studiju koja pomaže prilikom pisanja kôda dovršavajući započete izraze, ukazujući na propuste i slično.

Intermediate Language (IL) vidi MSIL.

IP-adresa *Internet Protocol* adresa je 32-bitni (4 bajta) broj koji jedinstveno predstavlja neko računalo spojeno na Internet, a služi za komunikaciju između računala. Više informacija u 1. poglavlju.

iznimke Greška koja se pojavljuje prilikom nedozvoljenog korištenja vrijednosti.

klase Nacrti za kreiranje objekata.

kolačići Male tekstualne datoteke na računalu posjetitelja u kojima web-poslužitelj zapisuje podatke o posjetitelju.

komandna linija Tekstualno sučelje u kojem računalu dajemo jednu po jednu naredbu upisivanjem imena naredbe i navođenjem određenih parametara.

kompajliranje Proces u kojem se iz kôda pisanog u programskom jeziku generira kôd za izvršavanje na računalu.

konfiguracijske datoteke Datoteke, najčešće XML ili tekstualne u kojima definiramo postavke aplikacije.

konkatenacija Slijedno spajanje znakovnih nizova. Primjerice, rezultat konkatenacije nizova “dobar”, “ “ i “dan” je “dobar dan”.

konstruktor Metoda koja se izvršava prilikom stvaranja objekta iz neke klase.

kontrole Klase koje sadrže funkcionalnosti koje se često koriste. Zahvaljujući njima nije potrebno svaku funkcionalnost programirati *od nule*.

link (hyperlink) Osnovni element web-stranica. Predstavlja vezu između dvije stranice i povezuje cijeli web u jednu cjelinu.

localhost Ime koje predstavlja trenutno računalo. IP adresa lokalnog računala je 127.0.0.1, a njemu upućeni podaci ne putuju preko Interneta.

Longhorn Slijedeća klijentska inačica operativnog sustava Windows koja će se bazirati na .NET Frameworku.

metoda Komad kôda koji odrađuje neku funkcionalnost. Svaka metoda ima svoje ime, a, prema potrebi, može sadržavati jedan ili više ulaznih i izlaznih parametara.

Microsoft Intermediate Language (MSIL) Međujezik u koji se naš kôd kompajlira kako bi ga CLR mogao izvršiti.

MSDN Library Službena Microsoftova dokumentacija za .NET Framework i još hrpu drugih stvari.

namespace Putanja pomoću koje možemo referencirati članove neke klase.

nasljeđivanje Čin kreiranja nove klase tako da se preuzmu karakteristike postojeće.

neupravljani kôd Kôd koji se ne izvršava unutar .NET Frameworka.

newsgrupe Forumi na Internetu namijenjeni diskusijama o određenoj temi. *Newsgrupa* se sastoji od niza članaka (*postova*) i odgovora. Svaka *newsgrupa* ima svoj naziv iz kojeg je vidljiva tematika same grupe.

Northwind Ogledna baza podataka koju možete naći u svim Microsoftovim alatima za baze, a koja se često koristi za primjere.

ODBC *Open Database Connectivity* je sučelje koje omogućava aplikacijama zasnovanima na Windowsima pristup bazama podataka.

piksel Osnovna grafička jedinica (točka) za prikaz nekog sadržaja ili slike na ekranu.

poslužitelj Računalo koje odgovara na zahtjeve klijenta. Primjerice, web server prima zahtjeve za web-stranicama i šalje ih klijentu.

pozadinski kôd Programski kôd koji se nalazi u zasebnoj datoteci da se ne bi miješao s opisom sučelja.

prekoračenje Čin kojim nakon nasljeđivanja klase radimo vlastitu funkcionalnost za nekog već postojećeg člana i na taj ga način *prekoračujemo*.

preopterećenje Stvaranje više članova iste klase koji imaju različite funkcionalnosti, no pod istim imenom.

projekt Termin koji se koristi u Visual Studiju za kreiranje *assemblyja*.

RGB *Red-Green-Blue* je model kojim se opisuju boje na računalnom zaslonu. Svaka se boja može predstaviti kao zbroj tri različite boje – crvene, zelene i plave – a njihov udio u konačnoj boji mjeri se brojem između 0 i 255.

server v. poslužitelj

sintaksa Skup pravila koji propisuje kako se pišu naredbe u nekom programskom jeziku.

SOAP *Simple Object Access Protocol*. Protokol kojim komuniciraju web-servisi.

Solution Explorer Pomoćni prozor koji nudi popis svih datoteka i referenci koje koristimo u projektu.

spremljena procedura Procedura spremljena na SQL Serveru koja vrši neku radnju nad bazom podataka. Brža je od običnih SQL upita, njom možemo upravljati preko ulaznih, a vrijednosti dobivati preko izlaznih parametara.

SQL Server Najjača Microsoftova baza podataka koja se koristi u brojnim okolnostima.

SQL *Structured Query Language* je standardni jezik koji se koristi za rad s relacijskim bazama podataka. Služi za dohvaćanje zapisa iz baze, njihovo mijenjanje, dodavanje novih, brisanje postojećih, itd.

svojstva Parametri kojima određujemo karakteristike kontrola i objekata.

tag Tzv. "naredba" u HTML kôdu koja služi za opisivanje nekog sadržaja.

TaskList Pomoćni prozor Visual Studija koji može služiti kao podsjetnik na stvari koje još treba učiniti.

TCP/IP *Transmission Control Protocol / Internet Protocol* je niz standardnih protokola za razmjenu podataka preko mreže. Predstavlja temeljni protokol za komunikaciju računala na Internetu.

Toolbox Pomoćni prozor Visual Studija u kojem se nalaze sve dostupne kontrole.

učahurivanje Koncept koji govori da je implementacija objekta (klase) u potpunosti neovisna o njegovom sučelju.

upravljani kôd Kôd koji se u potpunosti izvršava unutar .NET Frameworka.

Usenet v. newsgrupe

validacija Proces analiziranja podataka i utvrđivanja slažu li se s prethodno postavljenim zahtjevima.

varijable "Mjesta" u memoriji gdje spremamo određene vrijednosti kako bismo ih mogli kasnije u kôdu koristiti.

Visual Studio .NET Razvojna okolina za pisanje aplikacija na .NET Frameworku.

višeoblič̨je Mogućnost da objekti (klase) na različite načine implementiraju ista sučelja.

W3C *World Wide Web Consortium.* Organizacija koje se brine o razvoju i standardizaciji brojnih protokola vezanih uz Internet.

web-aplikacija Skup svih web-skripti na istom *siteu* ili u istoj mapi koje osim adrese dijele postavke, sesije, kolačiće i slično.

web-servis Standardiziran način komunikacije među aplikacijama preko Interneta.

Whidbey Kôdno ime sljedeće inačice Visual Studija .NET koja će biti uz bok .NET Frameworku 2.0.

WSDL *Web Services Description Language*. Jezik temeljen na XML-u namijenjen opisivanju karakteristika web-servisa.

XML *Extensible Markup Language*. Izuzetno fleksibilan jezik za opisivanje podataka koji se koristi u brojnim tehnologijama, uključujući i web-servise, prilikom pristupa bazama podataka i slično.

XPath Jezik kojim opisujemo kako locirati i procesirati elemente u XML-dokumentima.

XSLT *Extensible Stylesheet Language Transformations*. Tehnologija pomoću koje transformiramo strukturu XML-dokumenta u drugi XML-dokument (treba li reći – promijenjene strukture).