Uvod u C#

Ovo je prva lekcija programiranja u jeziku C#. Prvi deo vas uvodi u osnovna znanja koja su neophodna da bi ste se kasnije lako snašli. Ovde ćemo napraviti kratak pregled jezika C# i .NET okruženja, opis ovih tehnologija, prednosti njihove upotrebe, kao i međusobnu povezanost.

Počećemo sa osnovnim pitanjima vezanim za .NET okruženje. Ovo je nova tehnologija koja sadrži nove koncepte, sa kojima je isprva teško uhvatiti se u koštac, uglavnom zbog toga što predstavlja potpuno nov način rada u razvoju aplikacija. Kratak pregled osnova od velike je važnosti za razumevanje programiranja u jeziku C#.

Nakon kratkog pregleda preći ćemo na prost opis samog jezika C#, uključujući korene i sličnost sa jezikom C++.

Šta je .NET okruženje?

.NET okruženje je nova i revolucionarna platforma za razvoj aplikacija napravljena od strane Microsofta.

Primetićete da nismo rekli „razvoj aplikacija u operativnom sistemu Windows". lako prva verzija .NET okruženja radi na operativnom sistemu Windows, budući planovi podrazumevaju i rad na drugim sistemima, kao što su FreeBSD, Linux, Macintosh, pa čak i na uređajima klase ličnog digitalnog asistenta (PDA). Jedan od ključnih razloga za razvoj ove tehnologije je namera da ona postane sredstvo kojim se vrši integracija različitih operativnih sistema.

Pored toga, ova definicija .NET okruženja ne ograničava mogućnosti bilo kog tipa aplikacije. Ograničenja zapravo ne postoje - .NET okruženje daje vam mogućnost izrade Windows aplikacija, Web aplikacija, Web servisa i skoro svega ostalog što bi ste mogli zamisliti.
.NET okruženje dizajnirano je tako da se može koristiti iz bilo kog jezika: C#, C++, Visual Basic, JScript, pa čak i starije jezike kao što je COBOL. Da bi sve to funkcionisalo, pojavile su se i posebne verzije ovih jezika za .NET: Managed C++, Visual Basic .NET, JScript .NET, kao i razni drugi. Ne samo da svi oni imaju pristup .NET okruženju, već mogu i međusobno komunicirati. Sasvim je moguće kod programiranja u jeziku C# koristiti kod napisan u Visual Basic .NET-u, kao i obrnuto.

Sve ovo omogućuje dosad nezamisliv nivo višestruke namene, što pored ostalog čini .NET okruženje toliko atraktivnim.

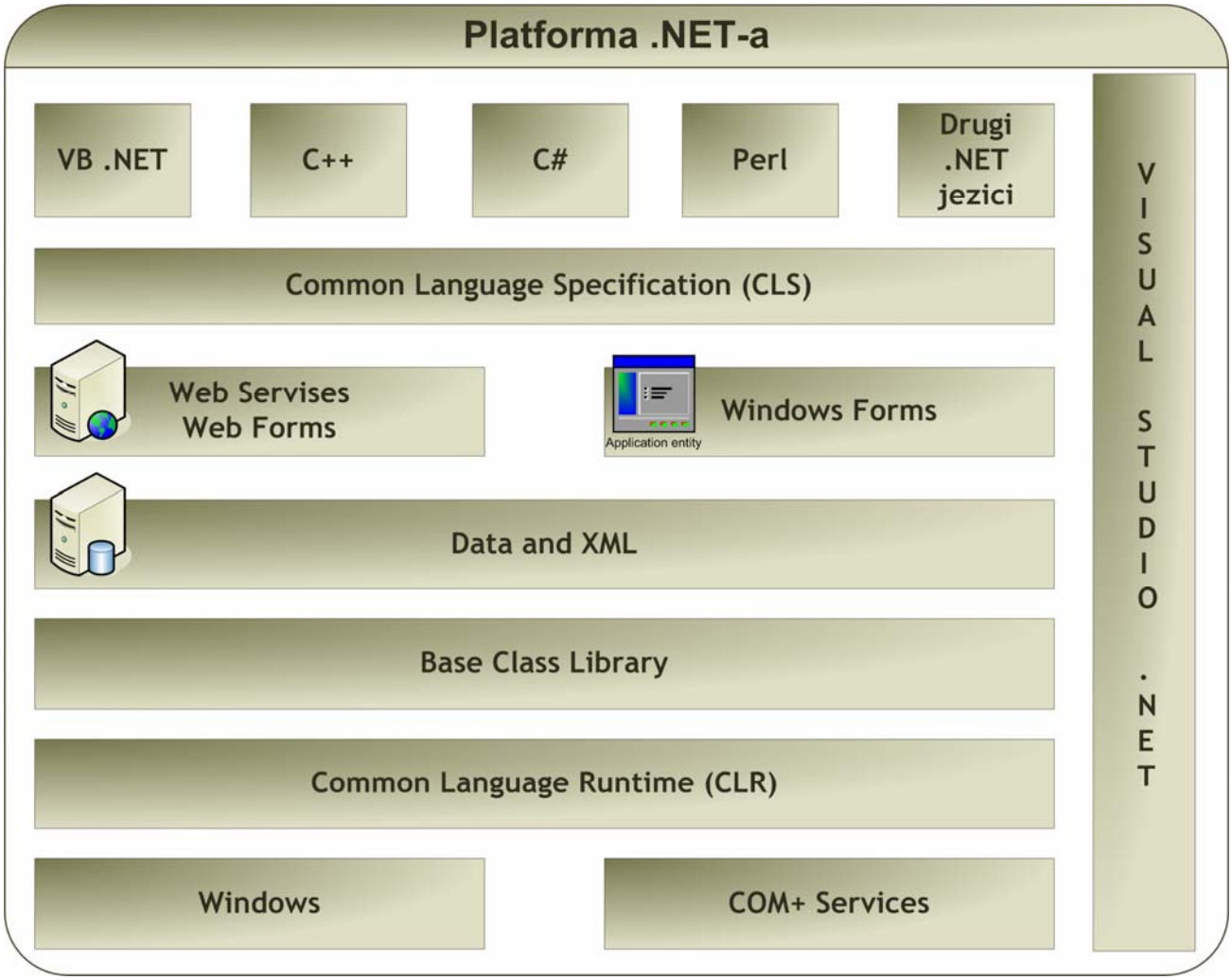
Šta se nalazi unutar .NET okruženja?

.NET okruženje se najvećim delom sastoji od ogromne biblioteke koda koju upotrebljavamo preko klijentskih jezika (kao što je C#) koristeći tehnike objektno orijentisanog programiranja (OOP). Ova biblioteka podeljena je na različite module koje koristimo u zavisnosti od zadatka. Na primer, jedan modul sadrži delove potrebne za pravljenje Windows aplikacija, drugi je vezan za mreže, a neki za razvoj Weba. Neki moduli su podeljeni u različite podmodule, kao što su moduli za pravljenje Web servisa unutar modula za razvoj Weba.

Namera je da različiti operativni sistemi podržavaju neke ili sve module zavisno od njihovih karakteristika. Na primer PDA će imati podršku za sve osnovne funkcionalnosti .NET modula, što se svakako ne odnosi i na one namenske ili rede korišćene.

Jedan deo biblioteke .NET okruženja definiše samo osnovne **tipove.** Tip je u stvari način na koji se predstavlja podatak (kao što je recimo označena 32-bitna celobrojna veličina), a definicijom fundamentalnih tipova podataka olakšava se interoperatibilnost izmedu jezika koji koriste .NET * okruženje. To se naziva **zajednički sistem tipova (engl.** *Common Type System* - **CTS).**

Okruženje osim biblioteke poseduje i .NET zajedničko **izvršno jezičko okruženje (engl.** *Common Language Runtime* - **CLR),** koje je odgovorno za izvršavanje svih aplikacija razvijenih uz pomoć :NET biblioteke.



Kako napisati aplikaciju koristeći .NET okruženje?

Pisanje aplikacija unutar .NET okruženja u stvari znači pisanje koda (koristi se bilo koji jezik koji podržava ovo okruženje) uz pomoć .-NET biblioteke koda. U ovoj semestru koristićemo VS za razvoj, koji je moćno i integrisano razvojno okruženje koje podržava jezik C# . Prednost ovog okruženja je lakoća kojom se mogućnosti .NET-a mogu integrisati u naš kod. Kod koji ćemo pisati biće u potpunosti C#, ali demo koristiti .NET okruženje, a takođe i dodatne alate VS-a kada zatreba.

Da bi se C# kod izvršio neophodno ga je prevesti u jezik koji operativni sistem na kome je aplikacija razume, poznat kao prirodni kod. Ovo prevođenje se još naziva i **kompajliranje** koda, funkcija koju obavlja **kompajler.** Unutar .NET okruženja ta operacija ima dve faze.

MSIL i JIT

Kada kompajliramo kod koristeći biblioteku .NET okruženja, ne pravimo odmah prirodni kod specifičan za operativni sistem. Umesto toga, kompajliramo u **Microsoftov posredni jezik** (engl. *Microsoft Intermediate Language* - MSIL). Ovaj kod nije određen ni jednim operativnim sistemom i nije namenjen samo za jezik C#. Ostali .NET jezici - na primer, Visual Basic .NET - takođe se kompajliraju u njega u svojoj prvoj fazi. VS će izvesti ovaj korak ukoliko ga koristimo pri razvoju C# aplikacija.

Da bi se program izvršio, očigledno je potrebno još rada. Za taj posao zadužen je kompajler **u pravo vreme** (engl. *Just-In-Time* - JIT), koji prevodi MSIL u prirodni kod specificiran od strane operativnog sistema i arhitekture same mašine na kojoj se program izvodi. Tek tada operativni sistem može izvršiti program. Pojam „u pravo vreme" označava da se MSIL kod kompajlira samo kada je to potrebno.

U prošlosti je bilo neophodno kompajlirati kod u vise različitih aplikacija zavisno od samog operativnog sistema i arhitekture samog procesora. Često je to bio oblik optimizacije (da

bi se recimo kod naterao da radi brže na AMD setu čipova), ali s vremena na vreme moglo je biti kritično (da aplikacije, na primer rade i pod Win9x i pod WinNT/2000 okruženjima). Sve je to sada nepotrebno, jer JIT kompajleri (kako im samo ime kaže) koriste MSIL kod, koji je nezavisan od mašine, operativnog sistema ili procesora. Postoji nekoliko JIT kompajlera - svaki za različitu arhitektura, a koristiće se odgovarajući da bi se napravio traženi prirodni kod.

Lepota je u tome što je mnogo manje rada potrebno sa naše strane. U stvari, možemo zaboraviti na detalje koji se odnose na sistemsku zavisnost i koncentrisati se na mnogo zanimljiviju funkcionalnost našeg koda.

Sklopovi

Kada kompajliramo aplikaciju, MSIL kod se smešta u posebne sklopove (engl. *assembly).* Takvi sklopovi sadrže i izvršne programske datoteke koje mogu biti startovane direktno iz Windowsa bez potrebe za nekim drugim programima (takve datoteke imaju .exe nastavak) i biblioteke koje koriste i druge aplikacije (koje imaju . dll nastavak).

Sklopovi pored MSIL sadrže i takozvane **meta** informacije (odnosno podatke o podacima sadržanim u sklopu, nazvane još i **metapodaci**), kao i opcione **resurse** (dodatne podatke koje koristi MSIL, kao što su zvukovi i slike). Metapodaci dozvoljavaju sklopovima da budu potpuno samoopisujući. Ne trebaju nam nikakve dodatne informacije da bi smo koristili sklopove. Podrazumeva se, naravno da izbegnemo situacije kao što su nedostavljanje traženih informacija sistemskom registru i si., što je inače često bio problem pri korišćenju drugih platformi.

To u stvari znači da je distribucija aplikacija često jednostavna koliko i kopiranje podataka u direktorijum na nekom udaljenom računaru. Kako nam nisu potrebne nikakve dodatne informacije o ciljnim sistemima, možemo odmah pokrenuti izvršnu datoteku iz ovog direktorijuma i (pod uslovom da je .NET CLR instaliran) polećemo.

Naravno da nećemo uvek želeti da uključimo sve što je potrebno za izvršenje jedne aplikacije na jednom mestu. Možda poželimo da napišemo kod koji će moći da koriste i druge aplikacije. U tom slučaju korisno je takav višenamenski kod smestiti na mesto koje mogu koristiti sve aplikacije U .NET okruženju to mesto se naziva globalni keš sklopova (engl. *Global Assembly Cache* - GAC). Smeštanje koda u ovaj keš je jednostavno - samo postavimo programski skup koji sadrži ovaj kod u direktorijum koji sadrži keš.

Upravljivi kod

Uloga CLR-a se ne zaustavlja onog trenutka kada smo kompajlirali kod u MSIL, niti kada JIT kompajler sve to prevede u prirodni kod. Kod napisan u .NET okruženju je upravljiv tokom svog izvršenja (ova faza se obično naziva izvršna). To znači da CLR vodi računa o našoj aplikaciji kroz upravljanje memorijom, obezbeđivanje sigurnosti, dozvoljava međujezičko otklanjanje grešaka i tako dalje. Suprotno ovome, aplikacije koje ne rade pod kontrolom CLR-a nazivaju se neupravljive, određeni jezici kao što je C++ mogu se koristiti za pisanje takvih aplikacija (na primer, da bi se pristupilo funkcijama niskog stepena operativnog sistema). Sve u svemu, u jeziku C# možemo napisati samo onaj kod koji radi pod upravljivim okruženjem. Mi demo iskoristiti razne mogućnosti upravljivosti CLR-a i dozvoliti .NET-u da se sam pozabavi interakcijom sa operativnim sistemom.

Sakupljanje otpada

Jedna od najvažnijih mogućnosti upravljivog koda jeste koncept sakupljanja otpada. Ovo je metod .NET-a koji se stara o tome da memorija koju je aplikacija koristila tokom svog izvršenja bude obavezno ispražnjena kada aplikacija nije vise u upotrebi. Pre pojave .NET-a to je najčešće bila briga samih programera, te je usled nekoliko malih grešaka memorija misteriozno gubila čitave blokove jer bila pogrešno pozivana. To je najčešće bilo praćeno usporavanjem vašeg računara i obaveznim padom sistema.

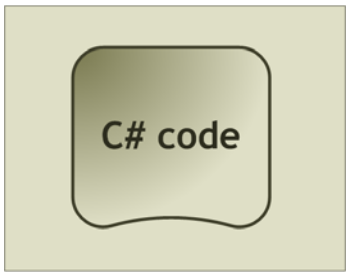
Sakupljanje otpada se odvija tako što učestalo ispituje memoriju vašeg računara i odbacuje sve što vam nije vise potrebno. Ono nema neki vremenski okvir izvršavanja, može se dogoditi hiljadu puta u sekundi, jednom u nekoliko sekundi, ili kad god, ali možete biti sigurni da će se dogoditi.

Kako je sakupljanje otpada nepredvidivo, morate ga imati na umu dok radite aplikacije. Kod koji za svoje izvršenje zahteva puno memorije morao bi iza sebe da počisti pre nego što dopusti sakupljanju otpada da to uradi za njega, ali to nije tako teško kako zvuči.

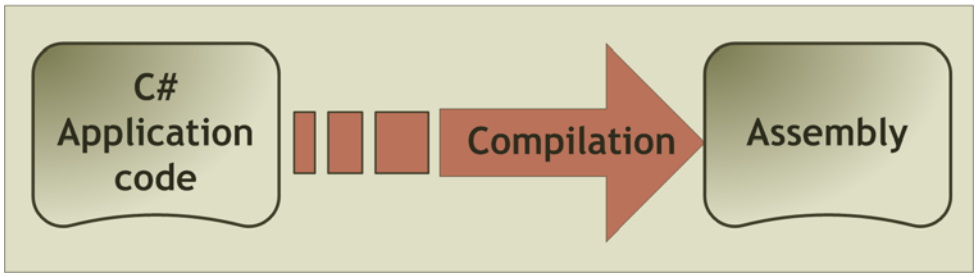
Uklapanje u celinu

Pre nego što nastavimo, sumirajmo sve korake koji su nam potrebni da bi smo napravili .NET aplikaciju.

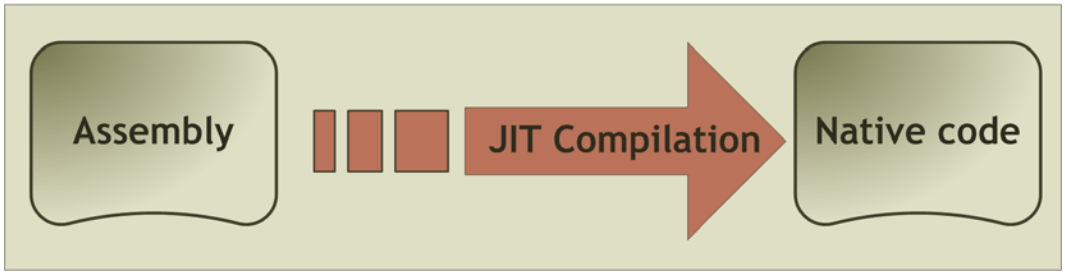
1. Kod za aplikaciju piše se korišćenjem .NET kompatibilnog jezika kao što je C#:



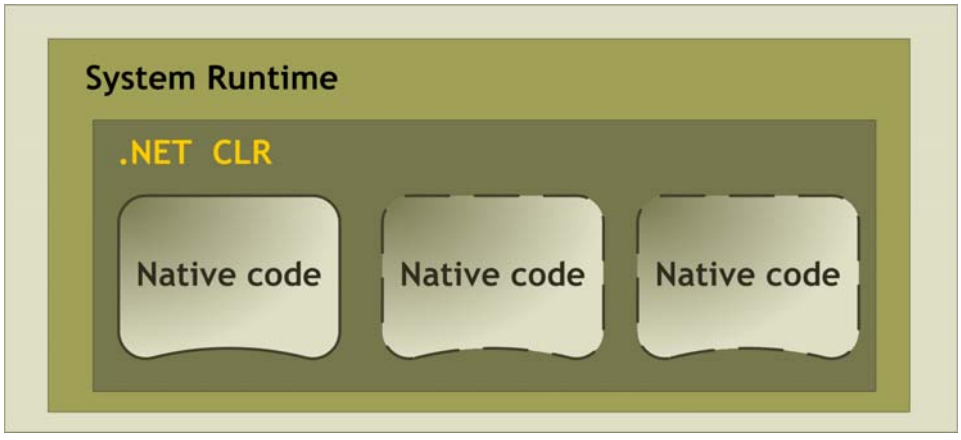
2. Ovaj kod se kompajlira u MSIL, koji je smešten u sklop:



3. Kod koji se izvršava (bez obzira da li je sam po sebi izvršni ili ga koristi neki drugi kod) mora se prethodno kompajlirati kroz JIT kompajler u prirodni kod:



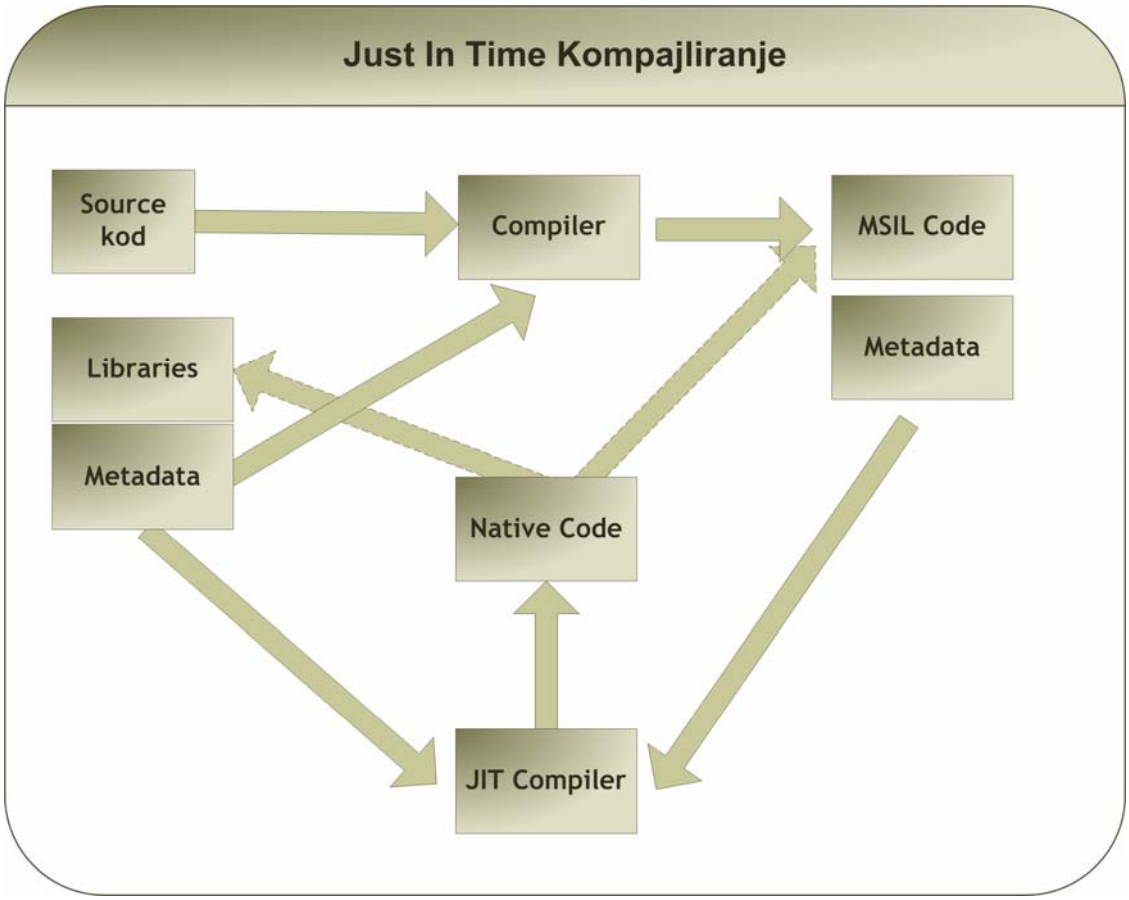
4. Prirodni kod se izvršava u kontekstu upravljanog CLR, zajedno sa bilo kojom drugom aplikacijom ili procesom:



Povezivanje

Postoji jedan dodatak gore pomenutom procesu. C# kod koji se kompajlira u MSIL u koraku 2 ne mora da bude smešten u jednu datoteku. Moguće je podeliti kod aplikacije u različite datoteke koji će se tokom kompajliranja objediniti u sklop. Ovaj proces je poznat i kao **povezivanje** i veoma je koristan. Ta podela se vrši jer je mnogo lakše raditi sa vise manjih datoteka nego sa jednom ogromnom. Možete izdeliti kod na logičke celine i smestiti ih u nezavisne datoteke na kojima se može nezavisno raditi i praktično zaboraviti na njih kada završite. Ovo nam olakšava pronalaženje određenih delova koda kada nam zatrebaju. Takođe omogućuje programerskim timovima da podele teret na manje delove kojima je lakše rukovati, kao i to da ukoliko dođe do provere nekog od tih delova ne postoji rizik da se ugroze celina programa ili njegovi delovi na kojima se radi.

Takođe celo MSJIT kompajliranje može se predstaviti i u celini što vam može pomoći da sagledate na drugi način:



Šta je C# ?

Kako je gore pomenuto, C# je jedan od jezika koji služi za izradu aplikacija koje mogu raditi pod .NET CLR-om. On predstavlja evoluciju jezika C i C++ koju je kreirao Microsoft da bi se radilo sa .NET okruženjem. U njegovom razvoju korišćene su mnoge prednosti drugih jezika, naravno uz otklanjanje njihovih mana.
Razvoj aplikacija u jeziku C# je jednostavniji nego u C++, jer je sintaksa prostija. Pa ipak, C# je moćan jezik i postoji veoma mali broj stvari koje bi smo uradili u C++ jeziku pre nego u C#-u. Paralelne mogućnosti jezika C# onima naprednijim u C++-U, kao što je direktno pristupanje i manipulisanje sistemskom memorijom, može biti izvedeno jedino kroz kod koji je obeležen kao **nesiguran.** Ove napredne tehnike programiranja su potencijalna opasnost (odakle im i ime) zato što je moguće pristupiti memorijskim blokovima koji su kritični za sistem, a sve to potencijalno vodi ka katastrofi. Iz ovog razloga, kao i drugih, ova tema nije obrađena ovim predavanjima.

S vremena na vreme kod u jeziku C# je malo razumljiviji od C++ koda. Jer je C# jezik **sigurniji** po pitanju dodele **tipa** promenljivim (za razliku od C++-a). To znači da ukoliko nekoj

promenljivoj ili nekom podatku dodelimo određeni tip, on ne može biti zamenjen nekim drugim nevezanim tipom. Postoje stroga pravila kojih se moramo pridržavati prilikom konverzija tipova, što u suštini znači da ćemo često morati da napišemo vise koda u jeziku C# nego u C++-U da bismo izvršili iste zadatke. Ali iz toga možemo izvući i neku korist. Kod je robusniji i jednostavnije je otkloniti greške, odnosno .NET u svakom trenutku može odrediti kojem tipu pripada traženi podatak.

Usled toga jezik C# nam ne omogućuje naredbe kao što su „Uzmi deo memorije počevši od 4. bajta unutar podatka i dužine 10 bajta i interpretiraj ga kao X", što ne mora da bude uvek loše.

C# je samo jedan od jezika koji omogućuju rad sa .NET razvojnim okruženjem, ali možda je i najbolji. Njegova prednost je u tome što je jedini napravljen od početka baš za .NET okruženje, pa samim rim može biti i vodeći jezik korišćen za verzije .NET-a namenjene dragim operativnim sistemima. VB.NET je, recimo, napravljen tako da bude što sličniji svojim prethodnicima i stoga ne može da iskoristi .NET biblioteku koda u potpunosti. Suprotno njemu, jezik C# je napravljen tako da može iskoristiti svaki deo biblioteke koda .NET okruženja koju ova može da ponudi.

Kakve aplikacije možemo pisati u C#-u?

Kako je gore navedeno, jezik C# nema ograničenja u pogledu toga kakve sve aplikacije možemo napraviti. C# koristi okruženje i samim tim nema ograničenja u vezi sa mogućim aplikacijama. Pa ipak, pogledajmo koji su tipovi najčešće pravljenih aplikacija:

- **Windows aplikacije** - To su recimo, aplikacije tipa Microsoft Office koje imaju izgled Windowsa i odlično se slažu sa njim. Ovo je uprošćeno korišćenjem modula **Windows formulara** unutar .NET okruženja, koji u stvari čini biblioteka k**ontrola** (kao što su dugmad, palete alatki, meniji i tako dalje) koje nam koriste pri izradi Windows korisničkog interfejsa (UI).
- **Web aplikacije** - Web strane koje možemo videti kroz bilo koji čitač. .NET okruženje pruža veoma moćan sistem generisanja Web sadržaja i to dinamički, dozvoljavajući personalizaciju, sigurnost i još mnogo toga. Ovaj sistem se naziva **Active Server Pages .NET (ASP.NET),** a jezik C# možemo koristiti za izradu ASP.NET aplikacija uz pomoć **Web formulara.**
- **Web servisi** - Predstavljaju nov i uzbudljiv način izrade raznovrsnih distribuiranih aplikacija. Koristeći Web servise preko Interneta možemo razmenjivati bilo koju vrstu podataka, koristeći prostu sintaksu, ne vodeći računa o tome u kom jeziku je napisana aplikacija niti na kom sistemu je postavljena.

Bilo kom od ovih tipova može zatrebati pristup bazama podataka, što se postiže korišćenjem **Active Data Objects .NET (ADO.NET)** odeljka .NET okruženja. Mnogi od ovih resursa mogu biti iskorišćeni, kao što su alati za izradu komponenata na mreži, iscrtavanje, izvođenje komplikovanih matematičkih zadataka i tako dalje.

Visual Studio.NET

U ovom predavanju za programiranje u jeziku C# koristimo Visual Studio .NET (VS), počev od aplikacija koje se pokreću iz komandne linije, pa sve do kompleksnih tipova projekata. VS nije neophodan za razvoj C# aplikacija, ali nam znatno olakšava stvari. Možemo (ako to poželimo) napisati naš kod u najobičnijem programu za uređivanje teksta, kao što je Notepad, i kompajlirati kod u sklopove koristeći kompajler iz komandne linije koji je u sastavu .NET okruženja. Međutim, zašto bismo to radili, kada možemo iskoristiti snagu VS-a da nam pomogne?

Sledi kratka lista razloga zbog kojih je VS pravi izbor za razvoj u .NET okruženju:

- VS automatizuje korake neophodne pri kompajliranju prirodnog koda, ali u isto vreme dozvoljava i kompletnu kontrolu nad opcijama ukoliko želimo da ih promenimo.
- VS program za uređivanje teksta je povezan sa jezicima koje podržava VS (uključujući jezik C#) i to tako da omogućuje pametnu detekciju grešaka i sugeriše odgovarajući kod na pravom mestu dok ga unosimo.
- VS sadrži i dizajnerske alate za aplikacije koje uključuju Windows formulare kao i Web formulare, koji vam omogućuju jednostavna tehnika povlačenja i puštanja (engl. *drag-and-drop)* dizajn elemenata korisničkog interfejsa.
- Mnogi tipovi projekata koji su mogući u jeziku C# mogu se napraviti sa, 'opštenamenskim' kodom koji se već nalazi na pravom mestu. Umesto da počinjemo od nule, često ćemo naći različite datoteke koje sadrže kod koji je za nas započet, smanjujući nam vreme provedeno u spremanju projekta.
- VS sadrži nekoliko čarobnjaka koji vam automatizuju zadatke opšte namene i koji mogu dodati odgovarajući kod postojećim datotekama, a da mi o tome ne vodimo brigu (ili u nekim slučajevima čak i da ne pamtimo tačnu sintaksu).
- VS sadrži moćne alatke za prikazivanje i navigaciju elemenata našeg projekta, bez obzira da li se radi o prirodnom kodu jezika C# ili nekoj slici, ili možda zvučnoj datoteci.

Pored toga što olakšava pisanje aplikacija, VS omogućuje i jednostavniju distribuciju i prilagođavanje projekata klijentima, kako bi ih oni lakše koristili ili instalirali.

VS omogućuje i napredne tehnike otklanjanja grešaka pri razvoju projekata, i to tako da možemo u isto vreme ući u kod jedne instrukcije dok pazimo u kakvom je stanju aplikacija.

Ima još mnogo toga, ali nadam se da ste shvatili o čemu se radi!

Rešenja u VS-u

Kada koristimo VS za razvoj aplikacija, to radimo kreiranjem **rešenja.** Rešenje, po VS terminologiji, predstavlja nešto vise od aplikacije. Rešenja sadrže **projekte,** koji mogu biti zasnovani na Windows formularima, Web formularima i tako dalje. Rešenja mogu sadržati i *vise projekata,* tako da možemo grupisati srodan kod na jedno mesto, iako će se oni kompajlirati u različite sklopove na različitim mestima na disku.

Ovo može biti od velike koristi, kada se radi na deljenom kodu (koji može biti smešten u globalni keš sklopova) u isto vreme dok aplikacije koriste taj kod. Otklanjanje grešaka u kodu je mnogo lakše ako se koristi jedinstveno razvojno okruženje, jer možemo pratiti instrukcije u višestrukim modulima koda.

Sažetak

U ovom predavanju upoznali smo se sa osnovnim pojmovima .NET razvojnog okruženja i sa mogućnostima pravljenja moćnih i raznovrsnih aplikacija. Saznali smo šta je neophodno da bi se kod napisan, recimo, u jeziku C# pretvorio u funkcionalnu aplikaciju, i koje su prednosti upravljanog koda koji radi pod opštim izvršnim jezikom .NET-a *(Common Language Runtime).*

Videli smo šta je u stvari jezik C#, kako je povezan sa .NET okruženjem i upoznali se sa alatom koji ćemo koristiti za programiranje u C# - Visual Studio .NET-u.

U sledećem predavanju naučićemo kako se C# kod piše pomoću VS-a, pa ćemo samim tim moći da se skoncentrišemo na jezik C# sam po sebi, umesto da brinemo o tome kako VS radi.

Pisanje programa u jeziku C#

Nakon što smo se upoznali sa tim šta je u stvari jezik C# i kako se uklapa u .NET okruženje, došao je trenutak da započnemo sa pisanjem koda. U tom poslu koristićemo Visual Studio .NET (VS), pa treba da naučimo osnovne stvari o torn razvojnom okruženju. VS je ogroman i komplikovan proizvod, tako da u prvo vreme korisniku može delovati zastrašujuće, ali uz njegovu pomoć pravljenje jednostavnih aplikacija može biti iznenađujuće prosto. Čim počnemo sa korišćenjem VS-a uvidećete da nije potrebno mnogo znati o njemu da bismo se poigrali sa C# kodom. Kasnije ćemo videti neke mnogo komplikovanije operacije koje je moguće izvesti u VS-u, ali za sada su dovoljna osnovna znanja.

Pošto zavirimo u VS, sastavićemo dve proste aplikacije. Za sada se nećemo mnogo opterećivati kodom, samo ćemo se upoznati sa tim koje su procedure potrebne da bi se aplikacija napravila i da bi funkcionisala kako treba, mada će sve to uskoro postati sporedno.

Prva aplikacija koju ćemo napraviti biće jednostavna **konzolna aplikacija.** Konzolne aplikacije ne koriste grafičko okruženje Windowsa, tako da ne moramo da brinemo o dugmadima, menijima, interakciji sa pokazivačem miša i tako dalje. Umesto toga, pokrenućemo aplikaciju iz komandne linije i komunicirati sa njom na mnogo prostiji način.

Druga aplikacija će biti bazirana na **Windows formularima.** Njen izgled, kao i rad sa njom, biće vrlo prepoznatljiv onima koji su koristili Windows i njeno kreiranje ne zahteva ništa vise rada nego prethodna. Ipak, sintaksa njenog koda je mnogo komplikovanija, mada često nećemo morati da vodimo računa o detaljima.

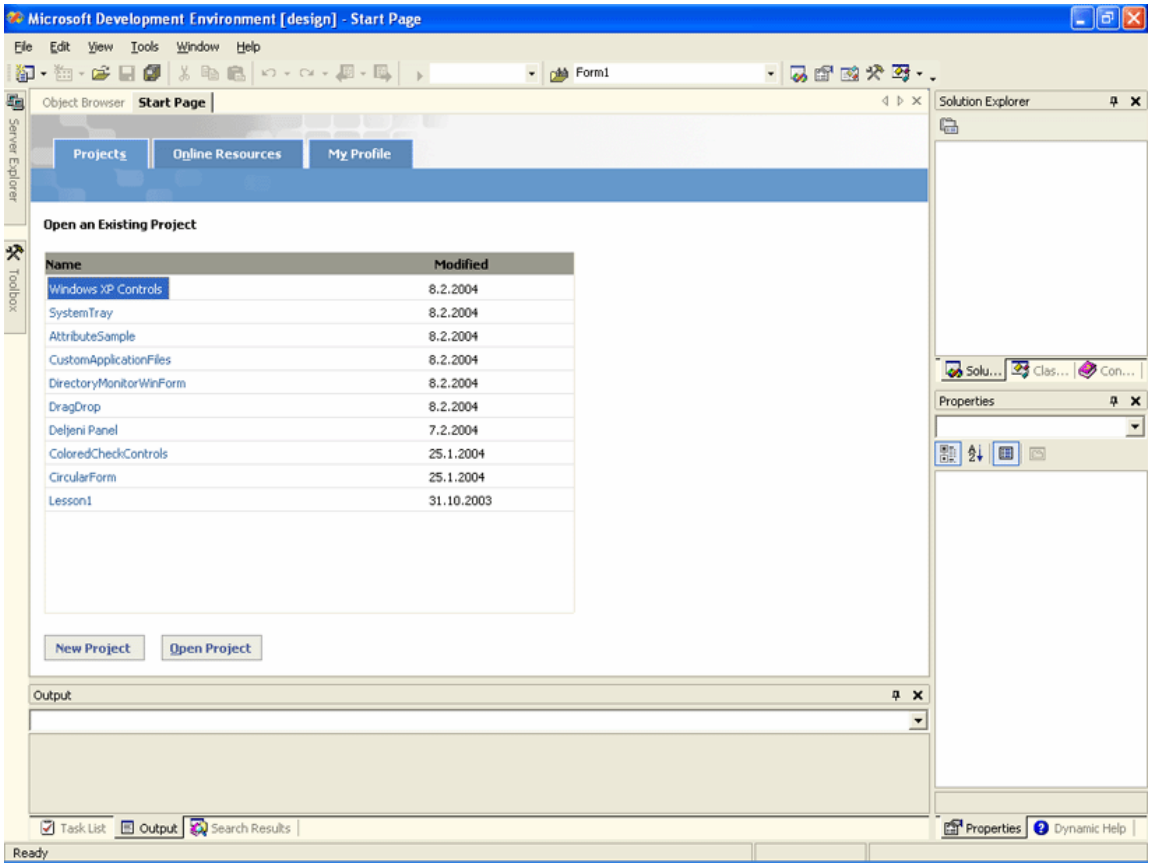
Dodatna fleksibilnost Windows aplikacija nije neophodna da bi se naučio C# jezik, dok jednostavnost konzolnih aplikacija omogućuje da se skoncentrišemo na učenje sintakse, a da za to vreme ne brinemo o tome kako aplikacija izgleda niti kakav je osećaj raditi sa njom.

Razvojno okruženje Visual Studio .NET

Kada prvi put učitate VS, odmah se pojavi nekoliko prozora, od kojih je većina prazna, zajedno sa nizom elemenata menija i ikona sa paletama alatki.

Ukoliko prvi put pokrećete VS biće vam predstavljena lista sa vrednostima namenjena za korisnike sa iskustvom u radu na prethodnim izdanjima ovog razvojnog okruženja. Automatski odabrani element! su sasvim u redu, za sada ih prihvatite jer tu nema ničega što se kasnije ne može promeniti.

Izgled VS okruženja je u potpunosti prilagodljiv, ali su nam sasvim dovoljne već postavljene vrednosti. One su sređene na sledeći način:



Glavni prozor, koji prikazuje uvodnu „početnu stranu" kada se VS pokrene, jeste onaj u kome se prikazuje ceo naš kod. Ovaj prozor je podeljen na kartice tako da možemo lako skakati sa jedne na drugu datoteku tako što ćemo pritisnuti mišem njihovo ime. Ovaj prozor ima i druge funkcije: može prikazati grafički korisnički interfejs koji dizajniramo za naš projekat, čiste tekstualne datoteke, HTML, kao i različite alate ugrađene u VS. Opisaćemo svaki, kako na njega naiđemo tokom rada.

Iznad glavnog prozora nalaze se palete sa alatkama, kao i VS meni. Ovde možete naći nekoliko različitih paleta sa alatkama rangiranih po funkcionalnosti - od alatki za snimanje i učitavanje datoteka, preko izgradnje i pokretanja projekta, do kontrola za otklanjanje grešaka. O svima njima ćemo pričati kada nam budu zatrebali.

Slede kratki opisi glavnih karakteristika VS-a, koje ćete najčešće koristiti:

- Server Explorer i Toolbox paleta sa alatkama izbacuju sopstvene menije kada se preko njih prede mišem, i otvaraju nam nove dodatne mogućnosti, kao što su pristup i podešavanje karakteristika i servisa samog servera, i pristup korisničkom interfejsu za izradu elemenata Windows aplikacija.
- Solution Explorer prozor prikazuje informacije o trenutno učitanim **rešenjima.** Rešenje je po terminologiji VS-a jedan ili vise projekata zajedno sa njihovom konfiguracijom. Ovde možemo videti različiti pogled na projekte unutar rešenja, kao što su datoteke iz kojih se sastoje i njihov sadržaj.
- Properties prozor omogućuje detaljniji pogled na sadržinu projekta i dozvoljava nam da izvedemo dodatno konfigurisanje pojedinih elemenata. Na primer, taj prozor možemo koristiti da promenimo izgled dugmeta unutar Windows formulara.
- Task List and Output prozor prikazuje informacije prilikom kompajliranja projekta zajedno sa zadacima koji treba da se izvrše (slično listi zadataka u Microsoft Outlooku). Ovi zadaci se mogu uneti ručno, a mogu biti i automatski generisani od VS-a.

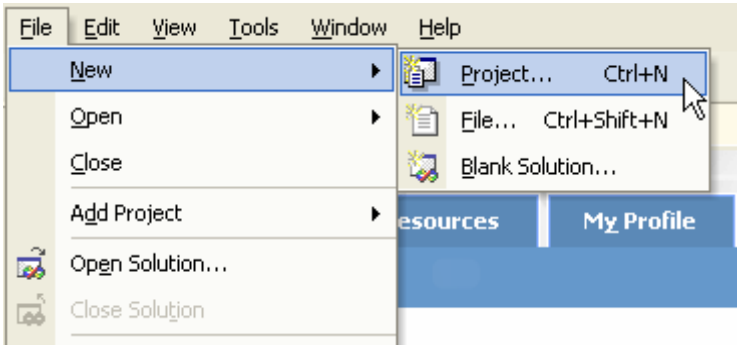
Počnimo sa pravljenjem prvog primera našeg projekta, koji će u sebe uključiti mnogo VS elemenata koje smo gore naveli.

Konzolna aplikacija

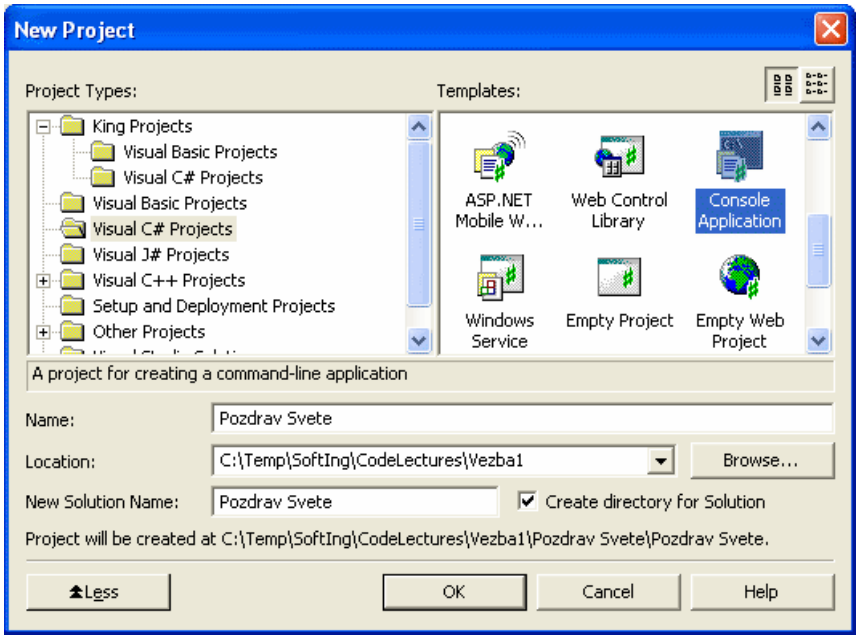
U ovom predavanju koristiti ćemo konzolne aplikacije, posebno na početku, stoga napravimo jednu jednostavnu.

Vežba br. 1.

Napravite novi projekat konzolne aplikacije birajući File | New | Project... iz menija:



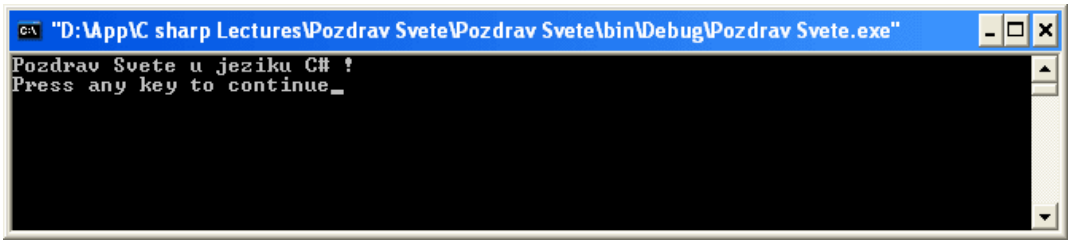
Izaberite Visual C# Projects direktorijum unutar prozora Project Types:, i tip projekta Console Application u okviru prozora Templates: (za ovo ćete morati malo da pomerite prozor na dole). U okviru za tekst Location: promenite putanju u **C:\Temp\SoftIng\LecturesCode\Vezba1** (ovaj će direktorijum biti automatski napravljen ukoliko već ne postoji), i ostavite podrazumevani tekst u okviru za tekst Name: **Pozdrav Svete**.



Pritisnite mišem na dugme OK.
Kada se projekat inicijalizuje dodajte sledeće redove u datoteku koja se bude pojavila u glavnom prozora:

```
1  using System;
2
3  namespace Pozdrav_Svete
4  {
5    /// <summary>
6    /// Summary description for Class1.
7    /// </summary>
8    class Class1
9    {
10     /// <summary>
11     /// The main entry point for the application.
12     /// </summary>
13     [STAThread]
14     static void Main(string[] args)
15     {
16       Console.WriteLine ("Pozdrav Svete u jeziku C# !");
17     }
18   }
19 }
20
```

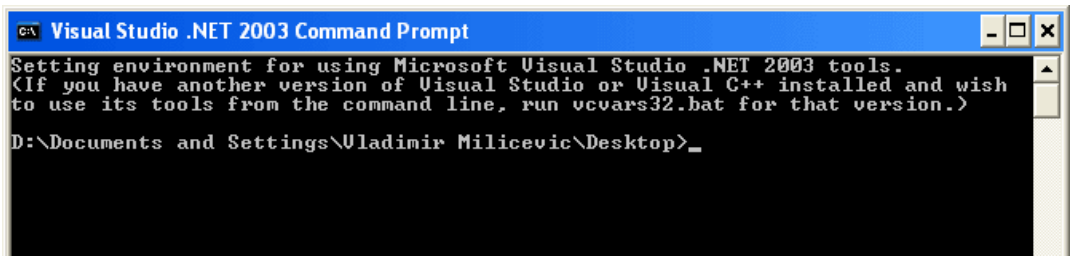
Izaberite **Debug | Start Without Debbuging** iz menija. Posle nekoliko trenutaka trebalo bi da vidite sledeće:



Pritisnite bilo koji taster da izađete iz aplikacije.

Takođe samu konzolnu aplikaciju možemo pokrenuti i na drugi način:

Odaberite **Start | Programs | Microsoft Visual Studio .NET 2003 | Visual Studio .NET Tools | Visual Studio .NET Command Prompt**. Pojaviće se komandni prozor i biće definisane sve promenljive okruženja potrebne prevodiocima i alatkama Visual Studija. Ako se pokrene obično komandno okruženje, bez zadavanja tih promenljivih, .NET alatke neće raditi.



Posle otvaranja komandnog prozora .NET-a uđite u folder u kome ste sačuvali ovu konzolnu aplikaciju. Unesite u komandnom prozoru sledeću komandu, koja prevodi kod konzolne aplikacije:

csc Class.cs <Enter>

Naravno programiranje pod celim Visual Studio .NET je tkz. Case Sensity, osetljiv velika i mala slova. Tako da morate da obratite pažnju pri kucanju koda. Ovde to nije slučaj, ali nije naodmet da se navikavate na velika i mala slova.

Prvo ćete dobiti obaveštenje o verziji prevodioca i izvršnog okruženja, a zatim i o autorskim pravima. Pokrenute program i bićete pozdravljeni na sledeći način:

C:\> Class1
Pozdrav svete na jeziku C# !

C:\>

Funkcija *Console.WriteLine*() dodaje tekstu indikator novog reda (engl. Newline character). U windowsu je to kombinacija znaka za početak reda i znaka za prelazak u novi red. Da biste u nekoliko iskaza pisali u istom redu, koristite funkciju *Write*(). Ona zahteva da sami dodate indikator novog reda. Sada u prethodnom programu zamenite iskaz *WriteLine()* sledećim iskazima:

```
Console.Write ("Zdravo svete ");
Console.Write ("na jeziku C#!\r\n");
```

Obrnuta kosa crta (\) ispred znaka je signal prevodiocu da je u pitanju izlazna sekvenca (engl. *escape sequence).* Iz sledeće tabele vidi se da C# koristi iste izlazne znake kao i C++.
Ako iza obrnute kose crte stavite neki dragi znak, dobićete od prevodioca poruku da nije prepoznao izlazni znak.

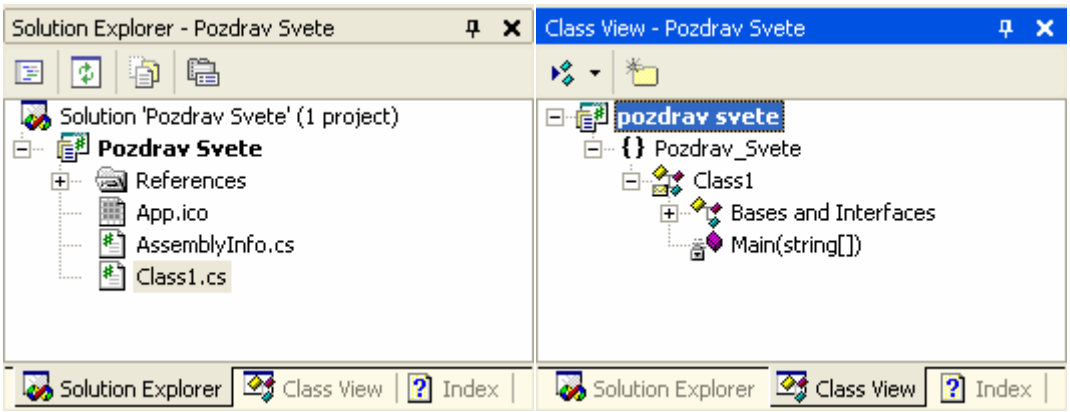
Izlazna sekvenca	Značenje
\a	Zvono. Zvučnik proizvodi kratak signal.
\b	Brisanje ulevo (engl. backspace). Pomera za jedno mesto ulevo.
\f	Kraj strane. Pomera na početak sledeće strane prilikom štampanja. Na ekranu daje kontrolni znak.
\n	Novi red. Pomera u sledeći red. Na ekranu se kursor pomera na početak sledećeg reda.
\r	Znak za povratak na početak reda. Pomera na početak tekućeg reda. Tekst koji se ispisuje na ekranu posle ovog znaka, pojavljivaće se preko tekućeg sadržaja reda.
\t	Horizontalan tabulator. Daje znak tabulatora.
\v	Vertikalan tabulator. Pomera kursor nadole za zadati broj redova. Na ekranu daje kontrolni znak.
\"	Navodnici. Bez obrnute kose crte se koristi kao graničnik za znakovne nizove. U kombinaciji sa obrnutom kosom crtom, prevodilac ga tumači kao običan znak.
****	Dvostruka obrnuta kosa crta. Ispisuje obrnutu kosu crtu.

Kako to *radi*

Za sada nećemo ispitivati kod koji smo koristili u ovom projektu; vise demo brinuti o tome kako koristiti VS da bi se kod osposobio za izvršavanje. Kao što ste videli, VS radi gomilu stvari za nas, što čini jednostavnim proces kompajliranja i izvršenja koda. Postoji vise načina za izvođenje čak i ovako jednostavnih koraka. Na primer, pravljenje novog projekta može se izvesti preko stavke menija **File| New | Project...** kao što smo i radili, ili pritiskanjem kombinacije tastera *Ctrl+Shift+N,* ili pritiskom mišem na odgovarajuću ikonu u paleti sa alatkama.
Slično tome vaš kod može biti kompajliran i izvršen na vise načina. Za metod koji smo koristili izborom stavke menija **Debug | Start Without Debbuging**, imamo takođe prečicu *(Ctrl+F5)*, a isto tako i ikonu u paleti sa alatkama. Kod možemo pokrenuti korišćenjem stavke menija Debug | Start (takođe pritiskajući *F5* ili odabirajući odgovarajuću ikonu; ili možemo kompajlirati kod bez njegovog pokretanja (sa uključenom ili isključenom proverom grešaka) koristeći **Build | Build**, *Ctrl+Shift+B,* ili odabirajući drugu ikonu. Kada je kod kompajliran možemo ga pokrenuti startovanjem dobijene .exe datoteke iz Windows Explorera, ili iz komandne linije.

Da bismo to izveli, moramo otvoriti komandnu liniju, preći u direktorijum **C:\Temp\SoftIng\LecturesCode\Vezba1\bin\Debug\,** uneti **Pozdrav Svete** i pritisnuti taster **Enter**.

U narednim primerima samo ćemo reći: „Napravite novi konzolni projekat", ili: „Pokrenite program", a vi izaberite način koji vam odgovara da biste to uradili. Ukoliko ne bude drugačije naglašeno, kod treba pokretati sa uključenim otklanjanjem grešaka.
Treba naglasiti da se poruka *Press any key to continue* koju ste videli u okviru konzolnog programa, pojavljuje samo ukoliko nije uključeno ispravljanje grešaka. Ako pokrenemo projekat u modu za otklanjanje grešaka prozor konzolnog programa će nestati. U principu to je dobro, ali ne i za prethodni primer. Ukoliko bismo tako uradili, ne bismo videli rezultate našeg rada.
Sada smo napravili projekat preko koga možemo detaljno pogledati neke delove ovog projektnog okruženja.
Prvi prozor na koji treba obratiti pažnju je Solution Explorer | Class View u gornjem desnom uglu ekrana koji je dole prikazan u oba režima rada (režim možete menjati pritiskajući dugmad u dnu prozora).

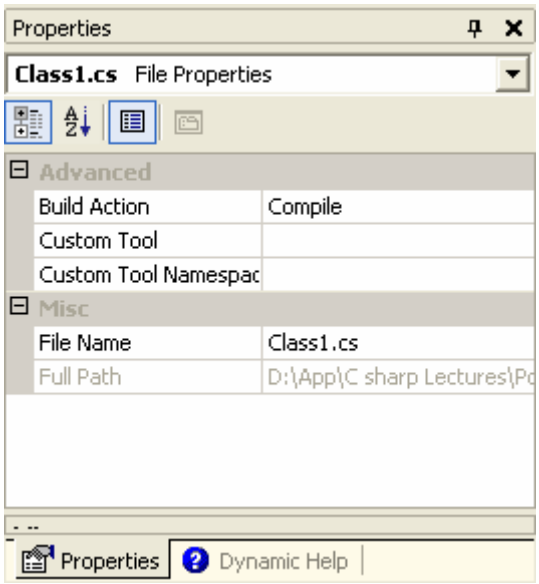


Ovaj pogled preko Solution Explorera prikazuje datoteke od kojih se sastoji projekat **Pozdrav svete**. Datoteka kojoj smo dodali kod je **Class1.cs** prikazana zajedno sa još jednom kod datotekom, **AssemblyInfo.cs**. (Sve datoteke sa C# kodom imaju nastavak **cs**). Ova druga datoteka sa kodom za sada nije naša briga, ona sadrži dodatne informacije o našem projektu koje nas se još uvek ne tiču.

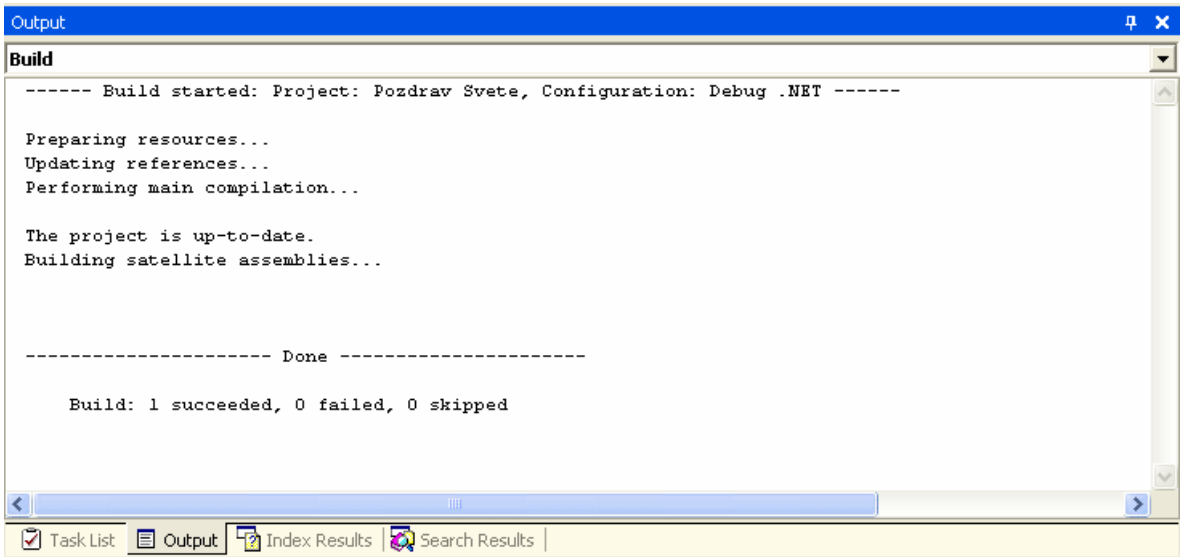
Unutar ovog prozora možemo menjati sadržaj glavnog prozora, koji će se kod unutar njega prikazivati, i to tako što ćemo otvoriti datoteku **.cs** koja nam je potrebna ili pritisnuti na nju desnim dugmetom miša i odabrati opciju View Code.
To možete uraditi i na drugi način, jednostavnim pritiskom mišem na dugme iz palete sa alatkama u gornjem delu prozora. U okviru ovog prozora možemo i manipulisati sa datotekama, brisati ih, menjati im ime i slično.

Ovde se takođe mogu nalaziti i drugi tipovi datoteka, kao što su resursne datoteke projekta (resursi su datoteke koje nisu uvek C# datoteke - to mogu biti i slike ili zvučne datoteke). Naravno, i sa njima možemo manipulisati preko istog interfejsa.

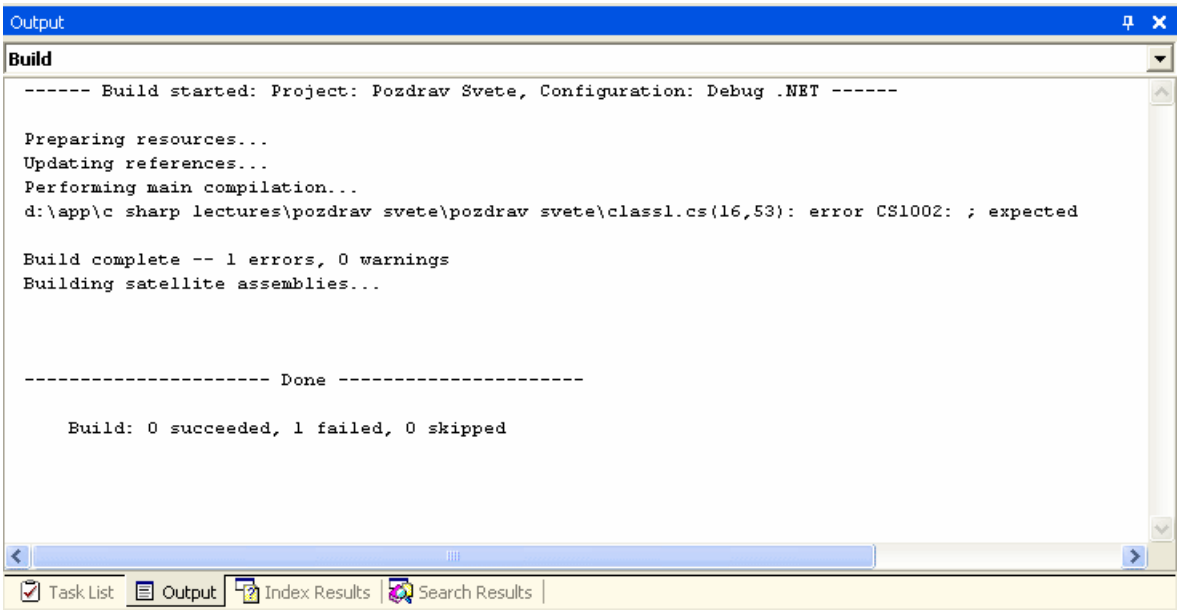
Lista References sadrži spisak .NET biblioteka koje koristimo u našem projektu. I to je nešto čime ćemo se pozabaviti kasnije, zato što su standardne reference za naš početak dovoljne.
Drugi pogled istog prozora, Class View, predstavlja način da pogledamo naš projekat kroz strukturu koda koji smo napravili. Tome ćemo se vratiti kasnije, a za sada odaberimo pogled Solution Explorer.
Primetićete da se sadržaj donjeg prozora menja u odnosu na to koju ikonu ili datoteku odaberete u gornjim prozorima. To je još jedan prozor koji ima vise pogleda, ali najvažniji od njih je pogled Properties:



Ovaj prozor pokazuje dodatne informacije o tome šta smo odabrali u prozora iznad njega. Na primer, kada izaberemo **Class1.cs** iz našeg projekta videćemo ono što je prikazano na slici. Ovaj prozor može da nam prikaže i informacije o drugim elementima koje odaberemo, kao što su komponente korisničkog interfejsa, što ćemo videti u drugom delu ovog predavanja koji se bavi Windows aplikacijama.
Često, ukoliko unesemo izmene u prozor Properties, to će direktno uticati na kod tako što će dodati nove redove koda ili menjati sadržaj naših datoteka. U nekim projektima provešćemo isto toliko vremena menjajući stvari u ovom prozora, koliko ćemo provesti u pisanju samog koda.
Sada pogledajmo prozor Output. Kada ste pokretali program verovatno ste primetili da se u ovom prozora pojavio neki tekst pre konzolnog prozora naše aplikacije. Na našem računara pojavilo se, na primer:



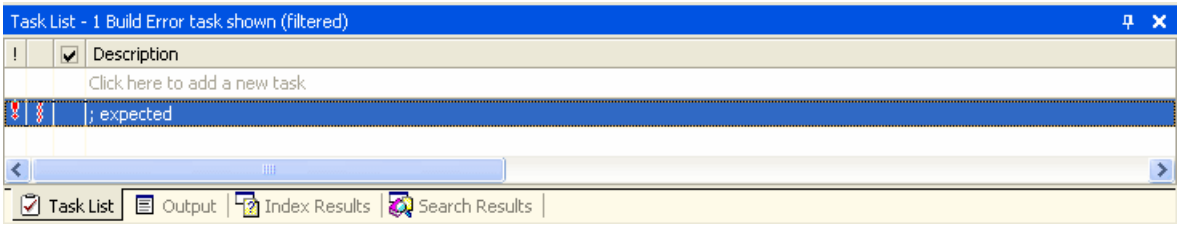
Kao što možete pretpostaviti, to je izveštaj o statusu datoteka koje se kompajliraju. Ovde ćemo dobiti i izveštaj o greškama koje se mogu dogoditi tokom kompajliranja. Na primer, probajte da obrišete tačku-zarez iz reda koji smo dodali unoseći kod u prethodnom koraku, i ponovo kompajlirajte. Ovaj put videćete sledeće:



Ovaj put projekat neće raditi.

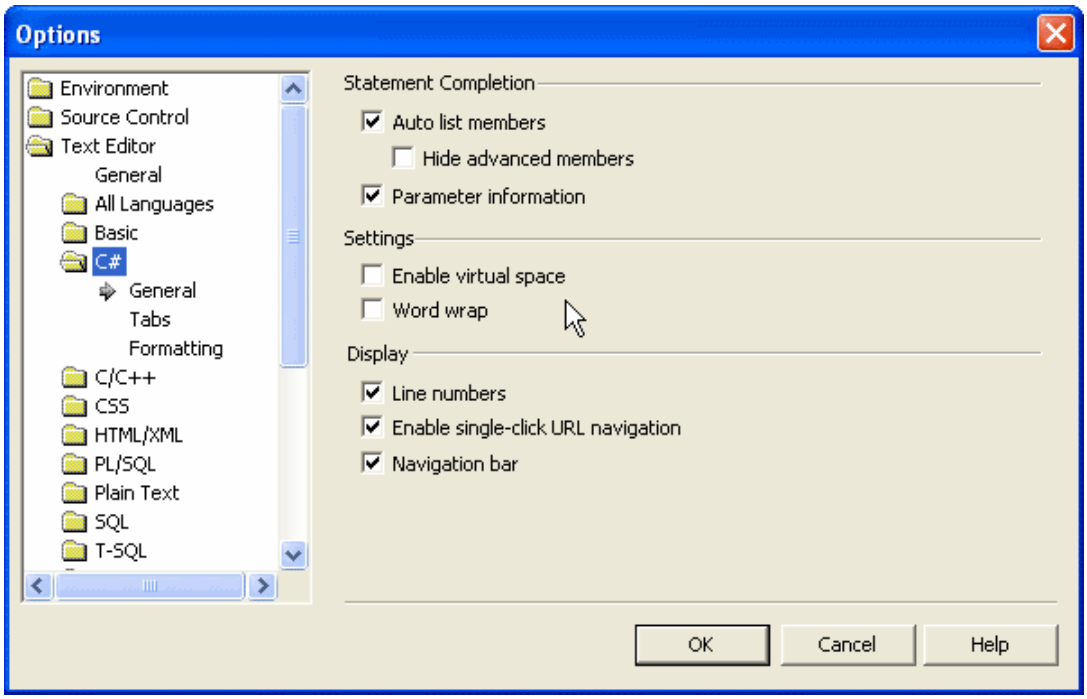
U sledećem predavanju kada budemo prešli na sintaksu C#-a, videćemo da se tačka-zarez očekuje kroz ceo kod - na kraju skoro svakog reda našeg koda.

Pošto sada moramo nešto uraditi da bismo naterali kod da radi, VS automatski dodaje zadatak u listu zadataka koja deli prostor sa prozorom **Output**:



Ovaj prozor će nam pomoći da utvrdimo gde se u našem kodu nalaze greške i vodiće računa o tome šta sve treba da uradimo da bi kompajlirali naš kod. Ako dva puta pritisnemo mišem na grešku prikazanu u ovom prozoru, kursor će se postaviti tačno na mestu gde je greška unutar koda (otvoriće se datoteka koda koja sadrži grešku ukoliko već nije otvorena), tako da problem možemo brzo rešiti. Takođe ćemo videti male crvene talasaste linije na mestu gde leži naša greška tako da kod nije teško pretražiti.

Zapazite kako je lokacija greške navedena kao broj reda. VS program za uređivanje teksta standardno ne prikazuje brojeve redova, a to je nešto što bi bilo od koristi da vidimo. Da bismo to postigli, trebalo bi potvrditi određena polja za potvrdu unutar dijaloga Options do koga se dolazi izborom stavke menija **Tools | Options...**. Ime polja za potvrdu je *Line Numbers* i nalazi se u kategoriji **Text Editor | C# | General**, kao što je prikazano na sledećoj slici:



Promenljive i izrazi

Računarski program je, najpreciznije rečeno, serija operacija koja manipuliše podacima. To važi i za najkomplikovanije primere kao što su obimne i višenamenske Windows aplikacije (Microsoft Office, na primer). Najčešće skriven od pogleda korisnika, taj skup operacija se uvek odvija u pozadini.
Ono što vidite na vašem monitoru toliko vam je poznato, da je teško zamisliti da je cela priča nešto vise od „pokretnih slika". U stvari, ono što vidite jeste samo predstava nekih podataka, čija gruba forma nije ništa drugo do niz nula i jedinica smeštenih negde u unutrašnjosti memorije vašeg računara. Sve što uradite na ekranu, bez obzira na to da li se radi o pokretanju pokazivača miša, pritiska na neku ikonu, ili unosa teksta u program za uređivanje teksta, rezultiraće naizmeničnom razmenom podataka u memoriji.
Naravno, postoje i manje apstraktne situacije koje vam mogu pokazati isto. Ukoliko uključite aplikaciju kalkulator, obezbedićete podatke u obliku brojeva i izvesti određene operacije sa njima na način na koji biste to uradili i na papiru, samo što je ovako mnogo brže.
Ako je računarski program u osnovi izvođenje operacija nad podacima, to znači da je nama potreban način na koji možemo uskladištiti neke podatke, kao i utvrđivanje metoda manipulacije njima. Ove dve funkcije se obezbeđuju preko **promenljivih** i **izraza** respektivno, a u ovom predavanju istražićemo šta to uopšteno i detaljno znači.
Prethodno je potrebno da se upoznamo sa osnovnom sintaksom jezika C#.

Osnove C# sintakse

Izgled i osećaj rada sa jezikom C# vrlo je sličan C++-u i Javi. U početku sintaksa izgleda prilično zbunjujuće, a i manje je nalik pisanom engleskom nego neki drugi jezici. Pa ipak, kako započnete sa programiranjem u jeziku C# shvatićete da je sintaksa razumljiva i da je moguće napisati vrlo jasan i čitljiv kod bez mnogo muke.

Za razliku od kompajlera za neke druge jezike, C# kompajler ne obraća pažnju na dodatne razmake unutar koda, bez obzira na to da li se sastoje od blanko znakova, tabulatora ili znakova za novi red. To znači da imamo prilično slobodu u formatiranju našeg koda, iako se pridržavanjem određenih pravila čitanje koda može znatno olakšati.
C# kod je sastavljen od niza iskaza, a svaki od njih se završava tačkom-zarezom. Kako se beli (prazan) prostor ignoriše, moguće je u jedan red staviti vise iskaza, ali zarad čitljivosti uobičajeno je preći u novi red nakon tačke-zareza. Na taj način izbegavamo vise iskaza u jednom redu. Savršeno je prihvatljivo (čak i prilično normalno) da se jedan iskaz unese u vise redova.
C# spada u jezike sa **blokovskom strukturom**, što znači da su svi iskazi delovi bloka unutar koda. Ovi blokovi, koji su oivičeni vitičastom zagradom ({i}), mogu da sadrže bilo koju vrstu iskaza, a i ne moraju ni jednu. Zapamtite da vitičaste zagrade ne zahtevaju tačku-zarez iza sebe.
Na primer, blok C# koda može izgledati ovako:

```
{
    <red koda 1,   iskaz 1 >;
    <red koda 2,   iskaz 2 >
            <red koda 3,   iskaz 3>;
}
```

Ovde naravno <red koda x, iskaz y> nisu pravi izgled C# koda - kroz ovaj tekst je samo prikazano gde bi trebalo staviti C# iskaze. Primetite da su u ovom slučaju red 2 i red 3 deo istog iskaza zbog toga što na kraju reda dva ne postoji tačka-zarez.

U ovom jednostavnom primeru način na koji je **uvučen** tekst tipičan je za C#. To nije izmišljeni metod pisanja C# koda, već ustaljena praksa; VS će sam uraditi sličnu stvar. Uopšteno, svaki blok koda ima svoj **stepen** uvlačenja, koliko će desno biti postavljen u odnosu na levu ivicu. Blokovi koda mogu biti **ugneždeni** jedan u drugi (odnosno, blokovi mogu sadržati druge blokove), i u torn slučaju se ugneždeni blokovi uvlače još više:

```
{
    <red koda 1>;
    {
        <red koda 2>;
        <red koda 3>;
    }
    <red koda  4>;
}
```

Takođe, redovi koda koji predstavljaju nastavak prethodnog reda uvlače se još vise, kao u prvom primeru u redu 3.
Zapamtite da ovaj stil pisanja ničim nije uslovljen. Međutim, ukoliko ga ne koristite, stvari mogu postati vrlo konfuzne!
Još jedna stvar koju često nalazimo u C# kodu jesu komentari. Komentari, strogo govoreći, uopšte nisu deo C# koda, ali na svu sreću postoje unutar njega. Komentari rade ono što im ime kazuje: omogućavaju dodavanje opisnog teksta vašem kodu - i to na srpskom (engleskom, francuskom, nemačkom, mongolskom i tako dalje) jeziku, a to će kompajler u potpunosti ignorisati. Kada se budemo bavili nekim delovima koda koji su duži, biće zgodno da dodamo podsetnike o tome šta radimo, na primer: „Ovaj red koda pita korisnika za broj", ili: „Ovaj deo koda je napisao Petar". C# sadrži dva načina da se to izvede. Možemo postaviti markere na početku i na kraju komentara, ili možemo upotrebiti marker koji označava da je sve do kraja ovog reda komentar. Ovaj poslednji metod je izuzetak od pravila koje smo gore pomenuli, da C# kompajler ignoriše prelazak u novi red.
Za obeležavanje komentara prvom metodom koristimo znakove **„ / * "** na početku komentara, i znakove **„ * / "** na kraju. Ovo možete koristiti za jedan red i za vise njih, ali u torn slučaju svi redovi izmedu markera postaju deo komentara. Jedina stvar koju ne možemo unositi unutar markera jeste **„ * / ",** jer bi to bilo shvaćeno kao kraj komentara.
Tako da je prihvatljivo sledeće:

```
/* Ovo je komentar  */
/*  Takođe...
    ...  i  ovo  */
```

Ali sledeće bi izazvalo probleme:

```
/* Komentari  se cesto  završavaju  " */"  znakovima  */
```

Ovde će kraj komentara (znakovi posle **„ * / "**) biti prepoznat kao C# kod i doći će do greške.
Drugi pristup komentarima uključuje započinjanje komentara znakom **„ / / ".** U nastavku možete unositi šta god vam je volja sve dok ste u istom redu! Sledeće ispravno:

```
// Ovo je druga  vrsta  komentara.
```

Ali, sledeće neće raditi, zato što će drugi red biti interpretiran kao C# kod:

```
// Takođe je i ovo,
    ali ovo nije.
```

Ova vrsta komentara je korisna za dokumentovanje iskaza, jer može biti postavljena u istom redu kao i iskaz:

<Iskaz>; // Objašnjenje iskaza

Treći tip komentara u jeziku C# dozvoljava vam da dokumentujete vaš kod. To su pojedinačni redovi komentara koji počinju sa tri **„ / "** simbola umesto sa dva, na primer:

/// Poseban komentar

Pod normalnim okolnostima kompajler ignoriše ovakve komentare kao i sve druge, ali VS možete konfigurisati tako da on pravi posebno formatirane tekstualne datoteke kada se projekat kompajlira, koje nam kasnije mogu koristiti da napravimo prateću dokumentaciju.

Vrlo je važno naglasiti da je C# kod osetljiv na upotrebu malih i velikih slova. Za razliku od nekih drugih jezika, kod se mora unositi pazeći na upotrebu velikih i malih slova. Ukoliko se toga ne pridržavamo, zaustavićemo kompajliranje projekta.

Osnove strukture C# konzolne aplikacije

Pogledajmo još jednom primer konzolne aplikacije iz prethodnog predavanja (Pozdrav Svete), i zavirimo u njegovu strukturu. Kod je izgledao ovako:

```
using System;
namespace Pozdrav_Svete
{
    /// <summary>
    /// Summary description for Class1.
    /// <summary>
    class Classl1
    {
        static void Main(string [] args)
        {
        //
        // TODO: Add code to start application here
        //
        Console.WriteLine("Pozdrav svete na jeziku C# !");
        }
    }
}
```

Možemo odmah uočiti sve elemente sintakse o kojima smo malopre pričali. Vidimo tačku-zarez, vitičaste zagrade, komentare zajedno sa odgovarajućim uvlačenjem.
Najvažniji deo koda za sada je:

```
static void Main (string []   args)
{
    //
    // TODO: Add code to start application here
    //
    Console.WriteLine("Pozdrav svete na jeziku C# !");
}
```

Ovaj deo koda se izvršava kada se konzolna aplikacija startuje, ili još preciznije, izvršava se blok koda oivičen vitičastim zagradama. Jedini red koda koji će nešto uraditi jeste onaj koji smo dodali automatski generisanom kodu - to je u stvari jedini red koji nije komentar. Ovaj kod jednostavno ispisuje neki tekst unutar konzolnog prozora, iako nas trenutno ne interesuje tačan mehanizam kojim se to izaziva.

Cilj nam je da u prvih nekoliko predavanja objasnimo osnove sintakse jezika C#, tako da nas ne interesuje kako u toku svog izvršavanja aplikacija dolazi do tačke gde se poziva **Console.WriteLine ().** Kasnije ćemo razjasniti značaj ovog dodatnog koda.

Promenljive

Kako je naznačeno u uvodu ovog predavanja, promenljive nam služe da u njih skladištimo podatke. U osnovi, o promenljivima unutar memorije računara razmišljamo kao o malim kutijama koje su poređane u nekoj polici. Te kutije možemo puniti raznim stvarima i prazniti ih, ili možemo samo zaviriti u njih.
Iako u osnovi svi računarski podaci u suštini sadrže istu stvar (nizove jedinica i nula), promenljive imaju vise tipova. Ponovo kroz analogiju sa kutijama možemo zamisliti da one dolaze u različitim veličinama i oblicima, tako da određene stvari mogu stati samo u određene kutije. Glavni razlog za ovakav način raspodele po tipovima leži u tome što postoje različiti tipovi podataka kojima se na drugačiji način manipuliše, tako da ograničavajući promenljive po tipu izbegavamo zabunu. Na primer, ne bi bilo logično tretirati na isti način nizove jedinica i nula koje zajedno čine digitalnu sliku kao i zvučne datoteke.
Da bismo koristili promenljive, one moraju biti deklarisane. To znači da moramo da im dodelimo ime i tip. Jednom kad ih deklarišemo, koristićemo ih za smeštanje onog tipa podataka koji je naznačen u deklaraciji.
Da bi se promenljiva navela u jeziku C# koristi se tip i ime kao u sledećem primeru:

<tip> <ime>;

Ako probamo da upotrebimo promenljivu koja nije deklarisana, kod se neće kompajlirati, ali u torn slučaju kompajler će vas tačno obavestiti šta je problem. Pored toga, ako pokušate da upotrebite promenljivu kojoj niste dodelili neku vrednost, to će takođe prouzrokovati grešku, ali kompajler će i to primetiti.
Činjenica je da postoji neograničen broj tipova koje možemo koristiti. To nam omogućava da sami definišemo svoje tipove ma koliko zamršeni oni bili.
Kada smo već kod toga, postoje određeni tipovi podataka koje će svi poželeti da koriste u određenom trenutku, na primer promenljive koje sadrže brojeve. Iz tog razloga postoje prosti unapred definisani tipovi promenljivih koje bi trebalo znati.

Prosti tipovi promenljivih

Prosti tipovi su recimo brojevne ili logičke vrednosti (tačno ili netačno) koje sačinjavaju osnovne blokove za izgradnju naših aplikacija, kao i za izgradnju drugih, složenijih tipova. Većina prostih tipova su brojevni, što nam u prvi mah izgleda malo čudno - zar nam ne treba samo jedan tip promenljivih da bismo uskladištili brojeve?
Razlog za podelu tipova brojevnih promenljivih leži u samom mehanizmu smeštanja brojeva kao nizova jedinica i nula u memoriju računara. Za celobrojne vrednosti jednostavno uzimamo određen broj bitova (nezavisnih cifri koje mogu biti jedinice ili nule) i predstavljamo naš broj u binarnom formatu. Promenljiva koja smešta N bitova dozvoljava nam da preko nje prikažemo bilo koji broj izmedu 0 i (2^n- 1). Bilo koji broj iznad ove vrednosti biće prevelik da se smesti u
promenljivu. Recimo da imamo promenljivu u koju možemo smestili dva bita. Preslikavanje izmedu celih brojeva i bitova koji ih predstavljaju izgleda ovako:

0=00
1=01
2-10
3-11

Ako želimo da smestimo vise brojeva, treba nam vise bitova (tri bita će nam recimo, omogućiti smeštanje brojeva od 0 do 7).
Nezaobilazan zaključak bi bio da nam za sve brojeve koje možemo zamisliti treba beskonačna količina bita, što baš i neće stati u naš verni PC. Čak i da postoji odgovarajući broj bitova za svaki broj, nikako ne bi bilo efikasno koristiti sve bitove za, recimo, promenljivu koja treba da smesti u sebe brojeve od 0 do 10 (ogroman prostor bi bio protraćen). Četiri bita bi nam ovde završila posao, dozvoljavajući nam da u isti memorijski prostor smestimo znatno vise vrednosti iz ovog opsega.
Umesto toga, imamo određen broj različitih celobrojnih tipova koje demo koristiti za različite opsege brojeva, čime ćemo zauzimati i različite količine memorije (sve do 64 bita). Sledi njihova lista:

Zapamtite da svaki od ovih tipova koristi jedan ili više standardnih tipova definisanih u .NET okruženju. Kao što smo razmotrili u prvom predavanju, korišćenje standardnih tipova nam omogućuje kombinovanje vise različitih jezika u jedan projekat ili rešenje. Nazivi koje koristimo za tipove u jeziku C# predstavljaju pseudonime za tipove koji su definisani u okruženju. Ova tabela je lista imena ovih tipova, i onoga šta predstavljaju unutar biblioteke .NET okruženja.

Tip	pseudonim za	dozvoljene vrednosti
sbyte	System.SByte	ceo broj između -128 i 127.
byte	System.Byte	ceo broj između 0 i 255.
short	System.Intl6	ceo broj između -32768 i 32767.
ushort	System.Uintl6	ceo broj O i 65535.
int	System.Int32	ceo broj između -2147.483.648 i 2147.483.647.
uint	System.UInt32	ceo broj između 0 i 4294967295.
long	System.Int64	ceo broj između -9223.372.036.854.775.808 i 9223.372.036.854.775.807.
ulong	System.UInt64	ceo broj između 0 i 1846.744.073.709.551.615.

Znak „u" koji stoji ispred imena nekih promenljivih jeste skraćenica za „unsigned" (neoznačeno), što znači da u taj tip promenljivih ne možemo smeštati negativne brojeve, što se može videti iz kolone dozvoljenih vrednosti u prethodnoj tabeli.

Pored celobrojnih, potreban nam je i način da smestimo brojeve sa **pokretnim zarezom**, odnosno decimalne brojeve. Postoje tri tipa promenljivih namenjenih brojevima sa pokretnim zarezom: **float, double i decimal**. Prva dva tipa čuvaju brojeve u pokretnom zarezu u obliku **+/- m x 2^e**, gde se dozvoljene vrednosti za m i e razlikuju za svaki tip. Tip decimal koristi oblik **+/- m x 10^e**. Ova tri tipa, zajedno sa dozvoljenim vrednostima, kao i realnim opsezima, prikazani su u sledećoj tabeli:

Tip	Pseudonim za	Minm	Maxm	Mine	Maxe	Pribl. min vrednost	Pribl. max vrednost
float	System. Single	0	224	-149	104	$1.5x10^{-45}$	$3.4x10^{38}$
double	System.Double	0	253	-1075	970	$5.0x10^{124}$	$1.7x10^{308}$
decimal	System.Decimal	0	296	-26	0	$1.0x10^{-28}$	$7.9x10^{28}$

Pored brojevnih tipova postoje još tri prosta tipa:

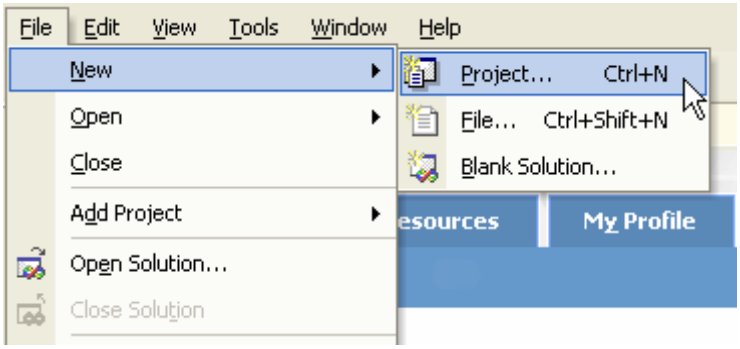
Tip	Pseudonim za	Dozvoljene vrednosti
char	System.Char	pojedinačan Unicode znak, smešten kao ceo broj između 0 i 65535.
bool	System.Boolean	logička vrednost, true ili false.
string	System.String	niz znakova.

Zapazite da ne postoji gornja granica za niz znakova koji sačinjavaju string. To je zbog toga što količina memorije koju on zauzima može biti promenljiva.
Logički tip bool je jedan od najčešće korišćenih tipova promenljivih u jeziku C#. Slični tipovi se najčešće koriste i u drugim jezicima. Vrlo je važno imati promenljivu koja može imati vrednosti true ili false, zbog logičkog toka aplikacije. Uzmite u razmatranje, na primer, na koliko pitanja može biti odgovoreno sa true ili false (odnosno, sa da ili ne). Poređenje vrednosti dve promenljive ili potvrđivanje nekog unosa podataka, predstavljaju samo dva od mnogih načina primene logičkih promenljivih koje ćemo uskoro isprobati.

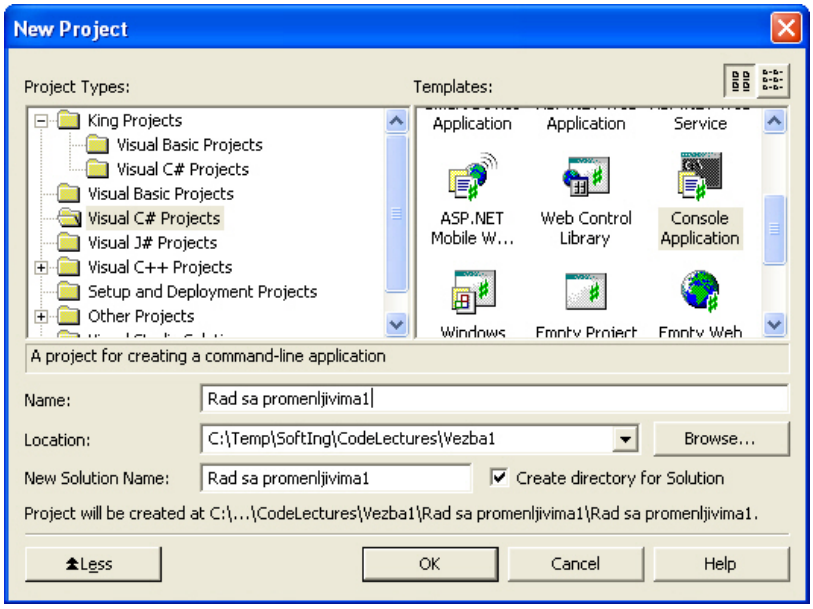
Pošto smo se upoznali sa tipovima promenljivih, uradimo kratak primer kako ih deklarisati i koristiti.

Vežba br. 2.

Napravite novi projekat konzolne aplikacije birajući File | New | Project... iz menija:



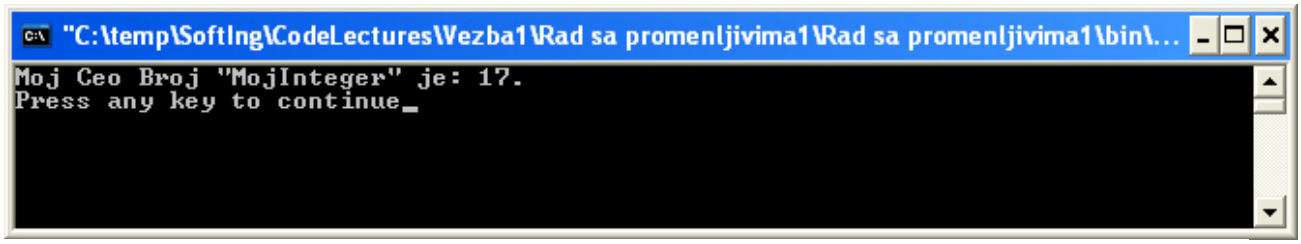
Izaberite Visual C# Projects direktorijum unutar prozora Project Types:, i tip projekta Console Application u okviru prozora Templates: (za ovo ćete morati malo da pomerite prozor na dole). U okviru za tekst Location: promenite putanju u **C:\Temp\SoftIng\LecturesCode\Vezba2** (ovaj će direktorijum biti automatski napravljen ukoliko već ne postoji), i ostavite podrazumevani tekst u okviru za tekst Name: **Rad sa promenljivima1.**



Dodajte sledeći kod u Classl.cs (obrišite redove sa komentarom na torn mestu):

```
14      static void Main(string[] args)
15      {
16        int MojInteger;
17        string MojString;
18        MojInteger = 17;
19        MojString = "Moj Ceo Broj \"MojInteger\" je:";
20        Console.WriteLine("{0} {1}.", MojString, MojInteger);
21      }
22    }
23  }
24
```

Pokrenite program (ukoliko vam nije isključen režim za otklanjanje grešaka, konzolna aplikacija će se zatvoriti pre nego što vidite šta se dogodilo):



Kako to radi

Kod koji ste dodali uradi sledeće:

- deklariše dve promenljive;
- dodeljuje vrednosti promenljivima;
- preko konzole ispisuje te dve vrednosti.

Promenljive se deklarišu u sledećem kodu:

```
int MojInteger;
string MojString;
```

Prvi red deklariše promenljivu tipa int i to pod imenom MojInteger, a dragi red deklariše promenljivu tipa string nazvanu MojString.

> *Zapamtite da je dodeljivanje imena promenljivima ograničeno, odnosno ne može baš sve biti ime promenljive. O ovome ćemo pričati u delu koji se bavi imenovanjem promenljivih.*

Sledeća dva reda koda deklarišu vrednosti:

```
MojInteger = 17;
MojString = "Moj Ceo Broj \"MojInteger\"  je:";
```

Ovde dodeljujemo dve fiksne vrednosti (poznatije kao vrednosti **literala** unutar koda) našim promenljivima koristeći **operator dodele** = (o operatorima ćemo detaljnije govoriti u delu koji se bavi izrazima u okviru ovog predavanja). Dodeljujemo vrednost 17 promenljivoj MojInteger, dok promenljivoj MojString dodeljujemo „MojInteger" is (uključujući i navodnike). Kada se dodeljuju znakovne vrednosti literala, zapamtite da moraju biti oivičene navodnicima. Da bi se to izvelo kako treba, neke znakove ne smemo koristiti unutar navodnika (na primer-same navodnike), ili ih moramo označiti tako što ćemo umesto njih koristiti neki niz znakova (poznat kao **escape** sekvenca) koji predstavlja one koje želimo da koristimo.
U ovom primeru koristili smo sekvencu „\" da bismo označili duple navodnike:

```
MojString = "Moj Ceo Broj \"MojInteger\"  je:";
```

Da nismo koristili escape sekvencu, i umesto nje probali:

```
MojString =  ""MojInteger"   je";
```

pri kompajliranju bi se pojavila greška.

Pri dodeljivanju znakova literala moramo biti oprezni sa prelazima u novi red - C# kompajler će odbaciti literale koji su veći od jednog reda. Ako želimo da proširimo literal na vise od jednog reda, moramo koristiti escape sekvencu \n. Na primer, sledeća dodela:

```
MojString =  "Ovaj string ima \nprelazaka u novi red.  ";
```

biće prikazana u dva reda u okviru konzolnog prozora.

Sve iskočne (engl. escape) sekvence sastoje se iz znaka obrnute kose crte (\ ) koji prati nekoliko znakova (videćemo kasnije koji su to znakovi). Pošto je znak obrnute kose crte iskorišćen u ovu svrhu, i za njega postoji escape sekvenca koja se jednostavno sastoji iz dva takva znaka, \\.

Vratimo se kodu. Ostao nam je samo još jedan red koji nismo proučili:

```
Console.WriteLine("{0}  {1}.", MojString, MojInteger);
```

Ovo je jako slično metodi ispisivanja teksta unutar konzole koju smo koristili u prvom primeru, ali smo sada naveli i naše promenljive. Ovo je tehnika koju ćemo koristiti u prvom delu knjige da bismo u okviru konzolnog prozora ispisali neki tekst. Unutar zagrada imamo dve stvari:

- string,
- listu promenljivih čije vrednosti želimo da ubacimo u izlazni string, odvojenu zarezima.

Čini se da string koji ispisujemo na izlazu, " {0} {1}. ", ne sadrži nikakav koristan tekst. Ali kada se program pokrene, vidimo da taj string u stvari predstavlja šablon u koji ubacujemo sadržaj naših promenljivih. Svaki znak oivičen vitičastim zagradama predstavlja mesto na koje se smešta sadržaj naših promenljivih. A svako mesto (ili string za formatiranje) predstavljeno je celobrojnom vrednošću oivičenom vitičastim zagradama. Celobrojne vrednosti počinju od nule i uvećavaju se za jedan, a ukupan broj mesta treba da bude jednak broju promenljivih koje su navedene u produžetku stringa i odvojene su zarezima. Kada se tekst ispiše na konzoli, svako mesto biva zamenjeno odgovarajućom vrednošću predviđene promenljive. U gore navedenom primeru { 0 } je zamenjena stvarnom vrednošću prve promenljive MojString, a {1} je zamenjeno sadržajem promenljive MojInteger.

Ovaj metod ispisa teksta na konzolu koristićemo i u narednim primerima.

Imenovanje promenljivih, konvencije i vrednosti literala

Kao što je spomenuto u prethodnom delu, za ime promenljive ne može se koristiti bilo koji niz znakova. I pored toga, ostaje nam dovoljno fleksibilan način imenovanja promenljivih.

Osnovna pravila pri imenovanju promenljivih su:

- Prvi znak imena promenljive mora biti ili slovo, ili znak podvlačenja (_), ili @.
- Naredni znakovi mogu biti slova, znakovi podvlačenja ili brojevi.

Kao dodatak tome postoje određene ključne reči koje označavaju neke radnje samom kompajleru, kao što su ključne reči using i namespace iz prethodnih primera. Ako kojim slučajem iskoristimo neku od ključnih reči kao ime promenljive, kompajler će se pobuniti, i ubrzo ćete shvatiti da ste napravili grešku, pa stoga ne brinite o tome mnogo.

Na primer, sledeća imena promenljivih su dobra:

```
myBigVar
VAR1
_test
```

Ova nisu:

```
99FlasaPiva
namespace
Sve-je-gotovo
```

Upamtite, C# je osetljiv na korišćenje velikih i malih slova, zbog čega moramo pamtiti tačan oblik imena koje smo dodelili promenljivoj. Čak i jedna greška ovog tipa sprečiće kompajliranje programa.

Posledica toga je i to da možemo imati više promenljivih koje se razlikuju samo u velikim i malim slovima. Na primer, ovo su sve različite promenljive:

```
myVariable
MyVariable
MYVARIABLE
```

Pravila imenovanja

Imena promenljivih su nešto sa čime ćete mnogo raditi. Zbog toga bi vredelo malo vremena utrošiti na neku opštu klasifikaciju i sortiranje imena koja ćemo koristiti. Pre nego što krenemo trebalo bi da imate na umu da je ovo kontroverzna oblast. Tokom godina različiti sistemi su dolazili i odlazili, a neki programeri se svojski bore da opravdaju sopstvene sisteme. Do skoro najpopularniji sistem bio je poznat kao mađarska notacija. U njemu svaka promenjiva sadrži prefiks koji se piše malim slovom i identifikuje tip. Na primer, ako je promenljiva tipa int onda stavljamo i (ili n) ispred njenog imena (recimo iGodiste). Koristeći ovaj sistem, lako se može utvrditi kog su tipa promenljive.

Primena tog sistema u modernijim jezicima kao što je C#, malo je nezgodna. Za one tipove promenljivih koje smo do sada upoznali, mogli bismo se lako snaći sa jednim ili dva slova kao prefiksima. Ali pošto sami možemo definisati kompleksnije tipove, a i kako postoji vise stotina kompleksnih tipova u osnovi .NET okruženja, taj sistem brzo postaje neupotrebljiv. Kako na jednom projektu može raditi vise ljudi, svaki od njih može za sebe pronaći posebnu

kombinaciju oznaka za prefikse i na taj način potpuno zbuniti ostale, što dovodi do pogubnih posledica.

Programeri su do sada shvatili da je najbolje imenovati promenljivu prema njenoj nameni. Ako iskrsne bilo kakva sumnja, dovoljno je lako pronaći kome tipu promenljiva pripada. U VS-u dovoljno je da pokazivač miša namestimo na ime promenljive kojoj želimo da otkrijemo tip i ispisaće se poruka koja nas o tome obaveštava.
Trenutno imamo dve konvencije prema kojima se promenljive imenuju u .NET okruženju, a to su **PascalCase** i **camelCase**.

Način na koji su napisana imena ove dve konvencije je prema njihovoj upotrebi: obe se odnose na to da imena promenljivih budu sastavljena iz više reči, a i naglašavaju da svaka reč počinje velikim slovom, dok su sva ostala mala. Konvencija camelCase ima dodatno pravilo - da prvo slovo kompletnog imena promenljive bude malo slovo.

Sledeći primer koristi camelCase konvenciju:

godište
imeUčenika
datumUpisa

A u sledećem je upotrebljena konvencija PascalCase:

Godište
PrezimeUčenika
ZimaNašegNezadovoljstva

Za proste promenljive pridržavaćemo se konvencije camelCase, dok demo PascalCase koristiti za naprednije imenovanje, što je u stvari preporuka Microsofta.

Konačno, vredi napomenuti da su sistemi imenovanja u prošlosti često koristili znak podvlačenja, obično da bi se odvojile reči unutar imena promenljivih, kao što je moja_prva_promenljiva. Ovakav pristup je sada odbačen.

Tabela 4.1. Prefiksi tipova podataka

Tip podataka	Prefiks	Primer – vrednosti
Boolean	**bln**	blnUlogovan
Byte	**byt**	bytGodiste
Char	**chr**	chrLozinka
Decimal	**dec**	decProdato
Double	**dbl**	dblRezultatIzracunavanja
Integer	**int**	intBrojacPetlje
Long	**lng**	lngIdKupca
Object	**obj**	objWord
Short	**sho**	shoTotalnaVrednost
String	**str**	strIme

Tabela 4.2. Prefiksi promenljivih po području važenja

Područje važenja	Prefiks	Primer – vrednosti
Global	**g**	g_strSnimiPutanju
Private to class	**m**	m_blnPromenaVrednosti
Nonstatic promenljiva, local to method	**(no prefix)**	

Vrednosti literala

U prethodnom primeru videli smo dve vrednosti literala - celobrojnu i znakovnu. I drugi tipovi promenljivih takođe mogu imati vrednosti literala, što će biti prikazano u sledećoj tabeli. Mnogi od njih uključuju sufikse, odnosno nizove znakova koji se dodaju na kraju literala da bi se naznačio željeni tip. Neki literali imaju više različitih tipova koje na osnovu konteksta određuje kompajler za vreme kompajliranja:

Tabela 4.3. Sufiksi literala kod tipova podataka

Tip(ovi)	Kategorija	Sufiks	Primer/dozvoljene vrednosti
bool	logička	nema	true ili false
int,uint, long, ulong	celobrojna	nema	100
uint,ulong	celobrojna	u ili U	100U
long, ulong	celobrojna	l ili L	100L
ulong	celobrojna	ul, uL, Ul,lu, U, Lu, ili LU	100UL
float	realna	f ili F	1.5F
double	realna	nema, d ili D	1.5
decimal	realna	m ili M	1.5M
char	znakovna	nema	'a', ili escape sekvenca
string	niz znakova	nema	"a..a", može se uključiti escape sekvenca

String literali

Ranije u ovom predavanju videli smo nekoliko iskočnih sekvenci koje možemo koristiti u literalnim stringovima. Vredi predstaviti kompletnu tabelu, da bismo imali podsetnik:

Tabela 4.4. Escape sekvence

Escape sekvenca	Znak koji pravi	Vrednost znaka u unicode-u
\ `	jednostruki navodnik	0x0027
\"	dvostruki navodnik	0x0022
\\	obrnuta kosa crta	0x005C
\0	nula	0x0000
\a	alarm(pravi pištanje)	0x0007

Tabela 4.4. Escape sekvence

Escape sekvenca	Znak koji pravi	Vrednost znaka u unicode-u
\b	unazad za jedan znak	0x00080
\f	prelazak na sledeću stranicu	0x000C
\n	novi red	0x000A
\r	prenos u novi red	0x000D
\t	horizontalni tabulator	0x0009
\v	vertikalni tabulator	0x000B

Kolona ove tabele koja prikazuje Unicode vrednost, predstavlja heksadecimalnu vrednost koju taj znak ima u Unicode skupu znakova.

Za svaki Unicode znak možemo navesti odgovarajuću iskočnu sekvencu, kao što su one u tabeli. Sekvenca se uvek sastoji od znaka \ iza koga sledi četvorocifrena heksadecimalna vrednost (na primer četiri cifre iza znaka **x** u gornjoj tabeli).

To znači da su sledeća dva stringa ekvivalentna:

```
"Petar\'s string."
"Petar\u0027s string"
```

Očigledno da imamo vise mogućnosti korišćenjem Unicode iskorišćenih sekvenci.

Takođe možemo deformisati i **literalne** stringove. To podrazumeva da su u taj niz uključeni svi znakovi oivičeni dvostrukim navodnikom, uključujući i one koji označavaju kraj reda ili iskočnu sekvencu. Jedini izuzetak je iskočna sekvenca za dvostruke navodnike koja se mora navesti da bi se izbegao prevremeni kraj niza. Da bi se ovo izbeglo, na početak niza znakova, pre navodnika, stavljamo znak @:

```
@"sinonim string literala."
```

Niz znakova, odnosno string, lako možemo deformisati na normalan način, ali sledeći zahteva ovaj metod:

```
@"Kratka  lista:
objekat 1
objekat 2"
```

Literalni stringovi naročito su korisni u imenima datoteka zato što ovi često koriste znak obrnute kose crte. Ako koristimo normalan string, moramo koristiti dvostruku obrnutu kosu crtu u celom stringu, na primer:

```
"C: \Temp\MojFolder\MojFajl.doc"
```

Sa litaralnim stringom, ovo može biti čitljivije. Sledeći literalni string predstavlja potpuno isto kao malopređašnji niz znakova:

```
@"C:\Temp\ MojFolder\MojFajl.doc"
```

Zapamtite da su nizovi znakova, odnosno stringovi, referentnog tipa, za razliku od onih koji su vrednosnog tipa Posledica toga je i to da se stringu može dodeliti vrednost null, što znači da string promenljiva ne pokazuje na string. Ovo ćemo kasnije detaljnije objasniti.

Određivanje i dodeljivanje vrednosti promenljivim

Setimo se da smo deklarisali promenljive koristeći samo njihov tip i ime, na primer:

```
int godište;
```

Potom smo joj dodelili vrednost uz pomoć operatora dodele =:

```
godište =  25;
```

Upamtite da promenljive moraju biti inicijalizovane pre nego što ih upotrebimo. Prethodni redovi mogu biti korišćeni kao inicijalizacija.

Ima još nekoliko stvari koje možemo uraditi, koje ćete vidati u C# kodu. Prvo, možemo navesti vise različitih promenljivih istog tipa u isto vreme, što je moguće izvesti ako ih odvojimo zarezom:

```
int xVelicina, yVelicina;
```

gde su i xVelicina i yVelicina obe navedene kao celobrojne promenljive.
Druga tehnika koju ćete sretati jeste dodeljivanje vrednosti promenljivima dok ih deklarišemo, što u stvari znači kombinovanje dva reda koda:

```
int  godište = 25;
```

A možemo i obe tehnike koristiti zajedno:

```
int xVelicina =4,   yVelicina = 5;
```

Ovde su xVelicina i yVelicina različitih vrednosti. Pogledajte sledeće:

```
int xVelicina, yVelicina = 4;
```

Ovo znači da je yVelicina inicijalizovana, a xVelicina samo deklarisana.

Izrazi

Videli smo kako deklarisati i inicijalizovati promenljive i vreme je da naučimo kako da sa njima manipulišemo. C# kod sadrži veći broj **operatora** koji tome služe, uključujući i operator dodeljivanja =, koji smo već koristili. Kombinovanjem operatora sa promenljivima i literalima (koji se nazivaju **operandi** kada se koriste zajedno sa operatorima) možemo formirati izraze, koji čine osnovne gradivne blokove pri računanju.

Postoji širok dijapazon operatora, od onih jednostavnih do najkomplikovanijih, koje ćete sretati samo u matematičkim primenama. Prosti uključuju sve osnovne matematičke operacije, kao što je; recimo operator + za sabiranje, a oni komplikovaniji, na primer, manipulaciju nad sadržajem promenljive i to prikazane u binarnom obliku. Postoje i logički operatori koji služe u radu sa logičkim promenljivima, kao i operatori dodele, recimo =.

U ovom predavanju skoncentrisaćemo se na matematičke i operatore dodeljivanja, dok ćemo za iduća ostaviti logičke operatore. Naime, preko logičkih ćemo ispitivati kontrolu toka programa.

Operatori se mogu grubo podeliti na tri kategorije:

- **unarni** operatori, koji rade samo sa jednim operandom;
- **binarni** operatori, koji rade sa dva operanda;
- **ternarni** operatori, koji rade sa tri operanda.

Najveći deo operatora spada u binarne, samo nekoliko u unarne, dok je ternarni samo jedan koji se još zove i **uslovni** operator (uslovni operator je logički, odnosno on vraća logičku vrednost, ali o tome u idućem predavanju).
Pogledajmo sada matematičke operatore, koji sadrže i binarnu i unarnu kategoriju.

Matematički operatori

Postoji pet prostih matematičkih operatora, od kojih dva imaju i unarnu i binarnu formu. U donjoj tabeli navedeni su svi ove operatori, zajedno sa kratkim primerom njihove upotrebe i rezultatima pri radu sa prostim numeričkim tipovima (celobrojnim i decimalnim):

Tabela 4.5. Operatori

Operator	Kategorija	Primer izraza	Rezultat
+	binarni	var1=var2+var3;	var1 je dodeljena vrednost sume var2 i var3
-	binarni	var1=var2-var3;	var1 je dodeljena vrednost razlike var2 i var3
*	binarni	var1 = var2 * var3;	var1 je dodeljena vrednost proizvoda var2 i var3
/	binarni	var1 = var2 / var3	var1 je dodeljena vrednost deljenja var2 i var3
%	binarni	var1 = var2 % var3	var1 je dodeljena vrednost koja je ostatak deljenja var2 i var3
+	unarni	var1 = +var2	var1 je dodeljena vrednost var2
-	unarni	var1 = -var2	var1 je dodeljena vrednost var2

Primeri su pokazani korišćenjem prostih numeričkih tipova. Ukoliko koristite druge proste tipove može doći do zabune. Sta mislite da će se dogoditi ako, na primer, saberete dve logičke promenljive ? U ovom slučaju ništa, zato što će se kompajler odmah pobuniti ako, recimo, pokušate da koristite operator + (ili bilo koji dragi matematički operator) sa bool promenljivama. Sabiranje dve promenljive tipa char takođe dovodi do malog nesporazuma. Zapamtite da se promenljive tipa char čuvaju kao brojevi, pa prema tome, sabiranje takve dve promenljive za rezultat daje broj (da budemo precizniji, broj tipa int). Ovo je primer **implicitne konverzije** i o njemu ćemo još dosta govoriti, kao i o **eksplicitnoj konverziji**, koja se takođe odnosi na slučajeve kada su var1, var2, i var3 različitih tipova.

Nakon ovoga, binarni + operator ima smisla koristiti sa promenljivama tipa string. U ovom slučaju tabela treba ovako da izgleda:

Tabela 4.6. Operatori

Operator	Kategorija	Primer izraza	Rezultat
+	binarni	var1=var2+var3;	var1 je dodeljena nadovezana vrednost stringova var2 i var3

Nijedna draga matematička operacija neće raditi sa stringovima.
Ostala dva operatora koje treba da upoznamo su operatori **inkrementiranja** i **dekrementiranja**. Oba su unarni operatori koji se koriste na dva različita načina: ili odmah ispred, ili odmah iza operanda. Pogledajmo na brzinu rezultate ovih prostih izraza, pa ih onda razmotrimo:

Tabela 4.7. Operatori inkrementiranja i dekrementiranja

Operator	Kategorija	Primer izraza	Rezultat
++	unarni	var1=++var2	var1 je dodeljena vrednost. var2+1, var2 je uvećana za 1.
--	unarni	var1=--var2	var1 je dodeljena vrednost. var2-1, var2 je umanjena za 1.
++	unarni	var1=var2++	var1 je dodeljena vrednost var2. var2 je uvećana za 1.
--	unarni	var1=var2--	var1 je dodeljena vrednost var2. var2 je umanjena za 1.

Ključna stvar kod ovih operatora je da se vrednost operanda uvek menja. ++ uvek uvećava svoj 1 operand za 1, a - ga uvek umanjuje za 1. Od toga gde se operator tačno nalazi, zavisi i rezultat koji je smešten u var1. Smeštanje operatora pre operanda, znači da se operand menja pre nego što se izvrše dalja izračunavanja. Ukoliko ih stavimo posle operanda, operand se menja tek posle izračunavanja.

Ovo zahteva još jedan primer. Razmotrite sledeći kod:

```
int var1, var2 = 5, var3=6;
var1 = var2++* --var3;
```

Pitanje je sledeće: koja će vrednost biti dodeljena promenljivoj var1 ? Pre nego što se izraz izračuna, operator -- će promeniti vrednost var3 iz 6 u 5. Operator ++ koji stoji posle var2, možemo ignorisati, jer se on neće izvršiti pre nego što se obavi računanje, tako da će var1 biti proizvod 5 i 5, odnosno 25.

Prosti unarni operatori mogu nam koristiti u mnogim situacijama. Oni su u stvari samo skraćenica za izraze tipa:

```
var1  =var1+ 1;
```

Ovaj tip izraza ima mnogo upotreba, pogotovo kada se radi o petljama.

Preko sledećeg primera videćemo kako da koristimo matematičke operatore i upoznaćemo se sa nekim novim, korisnim konceptima.

Vežba br. 3.

Napravite novi projekat konzolne aplikacije birajući File | New | Project... iz menija:

Izaberite Visual C# Projects direktorijum unutar prozora Project Types:, i tip projekta Console Application u okviru prozora Templates: (za ovo ćete morati malo da pomerite prozor na dole). U okviru za tekst Location: promenite putanju u **C:\Temp\SoftIng\LecturesCode\Vezba3** (ovaj će direktorijum biti automatski napravljen ukoliko već ne postoji), i ostavite podrazumevani tekst u okviru za tekst Name: **Promenljive2** Dodajte sledeći kod u Class1.cs:

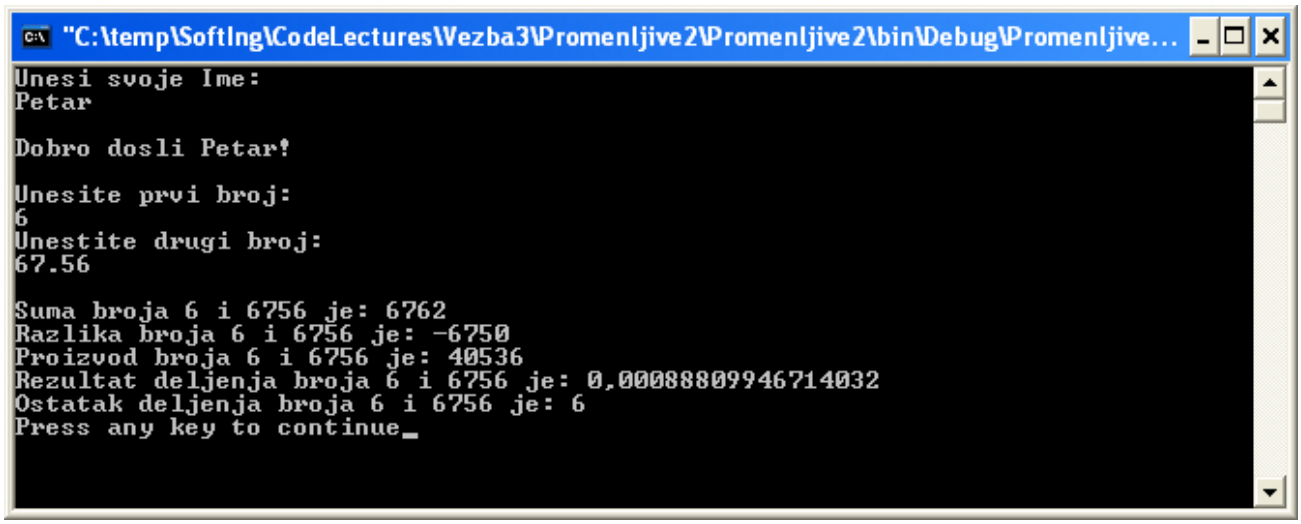
Operatori dodeljivanja

Do sada smo koristili prost operator dodele =, i možda vam je iznenađujuće da bilo koji drugi postoji. Međutim, ima ih vise i mogu biti vrlo korisni! Svi ostali operatori dodele rade na sličan način. Kao i kod operatora = i drugi se drže toga da se promenljiva kojoj se dodeljuje vrednost nalazi sa leve strane, dok se operandi i operatori nalaze sa desne strane. Kao i ranije videćemo tabelarni prikaz njihovih objašnjenja:

```
9     {
10      /// <summary>
11      /// The main entry point for the application.
12      /// </summary>
13      [STAThread]
14      static void Main(string[] args)
15      {
16        double prviBroj,drugiBroj;
17        string userName;
18        Console.WriteLine("Unesi svoje Ime:");
19        userName=Console.ReadLine();
20        Console.WriteLine("");
21        Console.WriteLine("Dobro došli {0}!", userName);
22        Console.WriteLine("");
23        Console.WriteLine("Unesite prvi broj: ");
24        prviBroj=Convert.ToDouble(Console.ReadLine());
25        Console.WriteLine("Unesite drugi broj: ");
26        drugiBroj=Convert.ToDouble(Console.ReadLine());
27        Console.WriteLine("");
28        Console.WriteLine("Suma broja {0} i {1} je: {2} ",
29          prviBroj,drugiBroj,prviBroj+drugiBroj);
30        Console.WriteLine("Razlika broja {0} i {1} je: {2} ",
31          prviBroj,drugiBroj,prviBroj-drugiBroj);
32        Console.WriteLine("Proizvod broja {0} i {1} je: {2} ",
33          prviBroj,drugiBroj,prviBroj*drugiBroj);
34        Console.WriteLine("Rezultat deljenja broja {0} i {1} je: {2} ",
35          prviBroj,drugiBroj,prviBroj/drugiBroj);
36        Console.WriteLine("Ostatak deljenja broja {0} i {1} je: {2} ",
37          prviBroj,drugiBroj,prviBroj%drugiBroj);
38
39      }
40    }
41  }
42
```

Snimite program, i pokrenite ga.

Unesite vaše ime i pritisnite Enter. Zatim unesite potrebne brojeve, i pritiskajte Enter.



Kako to radi

Osim što nam prikazuje operatore, ovaj kod nam donosi dva važna koncepta sa kojima ćemo se susresti mnogo puta u radnim primerima:

- Korisnički unos
- Konverzija tipa

Korisnički unos je sličan komandi **Console.WriteLine (),** koju smo već videli. Ovde, međutim, koristimo **Console.ReadLine().** Ova komanda traži od korisnika neki unos, koji smešta u promenljivu tipa string:

```
string userName;
Console.WriteLine("Unesi svoje Ime:");
userName=Console.ReadLine();
Console.WriteLine("");
Console.WriteLine("Dobro došli {0}!", userName);
```

Ovaj kod ispisuje sadržaj dodeljene promenljive, userName, pravo na ekran. U ovom primera učitavamo i dva broja. Ovo je malo kompleksnije, zato što Console.ReadLine() generiše string, a nama treba broj. Ovo nas uvodi u temu **konverzije tipa.** O ovome ćemo detaljnije u kasnije, za sada pogledajmo kod iz primera. Prvo, navodimo dve promenljive u koje treba smestiti brojeve koje unosimo:

```
double prviBroj,drugiBroj;
string userName;
```

Sledeće, ispisujemo poruku i koristimo komandu **Convert.ToDouble()** na promenljivoj tipa string koju nam je vratio red **Console.ReadLine(),** da bismo konvertovali promenljivu iz tipa string u tip double. Ovaj broj dodeljujemo promenljivoj **prviBroj** koju smo naveli:

```
Console.WriteLine("Unesite prvi broj: ");
prviBroj=Convert.ToDouble(Console.ReadLine());
```

Ova sintaksa je izuzetno prosta, a verovatno vas neće začuditi kada saznate da se mnoge druge konverzije odvijaju na sličan način. I drugi broj dobijamo na isti način:

```
Console.WriteLine("Unesite drugi broj: ");
drugiBroj=Convert.ToDouble(Console.ReadLine());
Console.WriteLine("");
```

Potom na izlaz šaljemo rezultate sabiranja, oduzimanja, množenja, deljenja, i na kraju prikazujemo ostatak posle deljenja, koristeći operator modula (%) za to:

```
Console.WriteLine("Suma broja {0} i {1} je: {2} ",
  prviBroj,drugiBroj,prviBroj+drugiBroj);
Console.WriteLine("Razlika broja {0} i {1} je: {2} ",
  prviBroj,drugiBroj,prviBroj-drugiBroj);
Console.WriteLine("Proizvod broja {0} i {1} je: {2} ",
  prviBroj,drugiBroj,prviBroj*drugiBroj);
Console.WriteLine("Rezultat deljenja broja {0} i {1} je: {2} ",
  prviBroj,drugiBroj,prviBroj/drugiBroj);
Console.WriteLine("Ostatak deljenja broja {0} i {1} je: {2} ",
  prviBroj,drugiBroj,prviBroj%drugiBroj);
```

Uočite da su izrazi prviBroj + drugiBroj i slično, parametri iskaza Console. WriteLine(), bez korišćenja neke posredne promenljive:

```
Console.WriteLine("Suma broja {0} i {1} je: {2} ",
  prviBroj,drugiBroj,prviBroj+drugiBroj);
Console.WriteLine("Razlika broja {0} i {1} je: {2} "
```

Ovakva sintaksa omogućuje veliku čitljivost koda, i smanjuje broj redova koje moramo da napišemo.

Operatori dodeljivanja

Do sada smo koristili prost operator dodele =, i možda vam je iznenađujuće da bilo koji drugi postoji. Međutim, ima ih vise i mogu biti vrlo korisni! Svi ostali operatori dodele rade na sličan način. Kao i kod operatora = i drugi se drže toga da se promenljiva kojoj se dodeljuje vrednost nalazi sa leve strane, dok se operandi i operatori nalaze sa desne strane. Kao i ranije videćemo tabelarni prikaz njihovih objašnjenja:

Tabela 4.8. Operatori dodeljivanja

Operator	Kategorija	Primer izraza	Rezultat
=	binarni	var1=var2;	var1 je dodeljena vrednost var2
+=	binarni	var1+=var2	var1 je dodeljena vrednost koja čini sumu var1 var2
-=	binarni	var1-=var2;	var1 je dodeljena vrednost razlike varl i var2
*=	binarni	var1*=var2;	var1 je dodeljena vrednost proizvoda var1 i var2
/=	binarni	var1/=var2;	var1 je dodeljena vrednost deljenja var1 i var2
%=	binarni	var1%=var2;	var1 je dodeljena vrednost ostatka deljenja var1 i var2

Kao što vidite, dodatni operatori uključuju var1 u izračunavanje, tako da:

```
var1  += var2;
```

daje isti rezultat kao i:

```
var1  = var1 + var2;
```

Upamtite da se operator += m*ože koristiti i sa stringovima, kao i* +.

Ukoliko koristimo dugačka imena za promenljive, ovi operatori nam mogu načiniti kod čitljivijim.

Prvenstvo operatora

Kada računar obrađuje izraz, on izvršava svaki operator prema redosledu. Međutim, to ne znači da se operatori uvek izračunavaju sleva nadesno.
Jednostavan primer je:

```
var1  = var2  + var3;
```

Ovde se prvo izvršava operator + pa tek onda =.
Postoje i druge situacije kada prvenstvo izvođenja operatora nije očigledno, na primer:

```
var1 = var2  + var3  * var4;
```

Ovde je redosled izvršenja sledeći: prvo operator * , zatim + i na kraju =. Ovo je standardni matematički redosled i daje isti rezultat kao da ste na papiru izveli neku operaciju u algebri. Kod takvih kalkulacija, možemo odrediti prvenstvo izvršavanja koristeći zagrade, na primer:

```
var1  =  (var2  +  var3)  *  var4;
```

Ovde se sadržaj unutar zagrada izračunava prvi, što znači da će operator + biti izvršen pre operatora *.

Redosled po kome se izvršavaju operatori prikazan je u tabeli dole - tamo gde su operatori istog prioriteta (kao recimo * i /) redosled se određuje sleva udesno:

Tabela 4.9. Prvenstvo operatora

Prvenstvo	Operator
Najviši	++, —(korišćeni kao prefiksi); +, -(unarni)
	*, /, %
	+, -
	=, *=, /=, %=, +=, -=
Najniži	++, — (korišćeni kao sufiksi)

Zapamtite da se redosled može promeniti upotrebom zagrada, kako je gore opisali

Imenovani prostori

Pre nego što nastavimo, bilo bi korisno da upoznamo imenovane prostore - metod .NET okruženja kojim se obezbeđuje kontejner koji sadrži aplikacioni kod, i to tako da i kod i njegov sadržaj mogu biti jedinstveno identifikovani. Imenovani prostori se takođe koriste i pri kategorisanju objekata .NET okruženja. Najveći deo tih objekata su definicije tipova, kao što su recimo prosti tipovi (System. Int32 i slično).
C# kod se podrazumevano sadrži u globalnim imenovanim prostorima. To znači da se tom kodu može pristupiti iz nekog drugog koda koji je unutar globalnih imenovanih prostora, tako što se navede njegovo ime. Možemo koristiti i ključnu reč namespace, da bismo eksplicitno definisali imenovani prostor u bloku koda oivičenom vitičastim zagradama. Takva imena moraju biti kvalifikovana, ukoliko će ih koristiti kod izvan imenovanog prostora.
Kvalifikovano ime je ono koje sadrži sve informacije o svojoj hijerarhiji. To znači da ukoliko imamo kod u jednom imenovanom prostoru, koji treba da koristi ime definisano u nekom drugom imenovanom prostoru, moramo uključiti referencu na ovaj imenovani prostor. Kvalifikovana imena koriste znak(.) između nivoa imenovanog prostora.
Na primer:

```
namespace LevelOne
{

    //kod u imenovanom prostoru LevelOne

    //ime "NameOne" definisano

}
    //kod u globalnom imenovanom prostoru
```

Ovaj kod definiše jedan imenovani prostor, LevelOne, kao i jedno ime u njemu, NameOne (ovde nije upotrebljeno ništa od pravog koda, da bi bilo uopšteno; umesto toga stavljeni su komentari tamo gde bi trebale da stoje definicije). Kod napisan unutar LevelOne imenovanog prostora, poziva ovo ime upotrebom "NameOne" - nije potrebna nikakva druga klasifikacija, Međutim, kod u globalnom imenovanom prostoru mora pozivati ovo ime na sledeći način: "LevelOne.NameOne".
U okviru imenovanih prostora možemo definisati ugneždene imenovane prostore, takođe koristeći ključnu reč namespace. Ugneždeni imenovani prostori se pozivaju preko svoje hijerarhije, opet koristeći tačku da bi naveli svaki nivo unutar hijerarhije. Ovo ćemo najbolje prikazati preko primera.

Razmotrite sledeće imenovane prostore:

```
namespace LevelOne
{
    //kod u imenovanom prostoru LevelOne

    namespace LevelTwo
    {
        //kod u imenovanom prostoru LevelOne.LevelTwo

        //ime "NameTwo" definisano
    }
}
//kod u globalnom imenovanom prostoru
```

Ovde se NameTwo mora pozivati kao "**LevelOne.LevelTwo.NameTwo**" iz globalnog imenovanog prostora; ili kao "**LevelTwo.NameTwo**" iz imenovanog prostora LevelOne; ili kao "NameTwo" iz imenovanog prostora **LevelOne.LevelTwo**.
Važno je naglasiti da su imena jedinstveno definisana od strane njihovih imenovanih prostora. Mogli bismo definisati ime "NameThree" u imenovanim prostorima LevelOne i LevelTwo:

```
namespace LevelOne
{
    //ime "NameThree" definisano

    namespace LevelTwo
    {
        //ime "NameThree" definisano
    }
}
```

Ovde smo definisali dva imena, LevelOne.NameThree i LevelOne.LevelTwo.NameThree, koja se mogu koristiti nezavisno jedno od drugoga.

Kada definišemo imenovani prostor, možemo pojednostaviti pristup imenima koje sadrže pomoću iskaza using. Iskaz using znači: „U redu, trebaće nam imena iz ovog imenovanog prostora, pa se nećemo mučiti da svaki put navodimo njihova puna imena". Na primer, u sledećem kodu kažemo da iz imenovanog prostora LevelOne treba da imamo pristup imenima u imenovanom prostora LevelOne.LevelTwo bez klasifikacije:

```
namespace LevelOne
{
    using LevelTwo;

    namespace LevelTwo
    {
        //ime

        "LevelTwo" definisano
    }
}
```

Sada iz imenovanog prostora LevelOne možemo pozvati ime LevelTwo.NameTwo pomoću "NameTwo".

Ponekad, kao u primera NameThree, ovo može dovesti do problema sa kolizijom istih imena u različitim imenovanim prostorima (kada se kod verovatno neće kompajlirati). U takvim slučajevima, možemo definisati **sinonim** za imenovani prostor kao deo iskaza using:

```
namespace LevelOne
{
    using LT=LevelTwo;
    // ime "NameThree" definisano
    namespace LevelTwo
    {
        // ime "NameThree" definisano
    }
}
```

Ovde se iz imenovanog prostora LevelOne ime LevelOne. NameThree poziva kao "NameThree", a ime LevelOne.LevelTwo.NameThree kao "LT.NameThree". Iskaz using odnosi se na imenovani prostor u kome se sadrži, kao i na sve imenovane prostore ugneždene u njega. U gore navedenom kodu globalni imenovani prostor ne može koristiti "LT.NameThree". Međim, koristiće iskaz using, na sledeći način:

```
using LT = LevelOne.LevelTwo;

namespace LevelOne
{
   //ime "NameThree" definisano
   namespace LevelTwo
   {
       //ime "NameThree" definisano
   }
}
```

Tada kod u globalnom imenovanom prostora kao i u LevelOne mogu koristiti "LT.NameThree".

Ovde je važno podvući još nešto. Iskaz using sam po sebi ne daje vam pristup imenima u drugom imenovanom prostoru. Ukoliko kod nije na neki način povezan sa vašim projektom tako što je definisan u izvornoj datoteci vašeg projekta, ili ukoliko nije deformisan unutar nekog drugog koda koji je povezan sa vašim projektom, nećemo imati pristup imenima koja sadrži. Isto tako, ako je kod koji sadrži imenovani prostor povezan sa vašim projektom, pristup imenima koje sadrži je omogućen, bez obzira na to da li koristimo using. Iskaz using nam prosto olakšava pristup ovim imenima, a može i prilično skratiti inače dugačak kod, da bi ga napravio razložnijim.

Ukoliko se vratimo na kod u *Pozdrav Svete*, sa početka ovog predavanja, videćemo sledeće redove koji se odnose na imenovane prostore:

```
using System;

namespace Pozdrav_Svete
{
.....
.....
}
```

Prvi red koristi using da definiše imenovani prostor System koji će biti korišćen u ovom C# kodu, i može mu se pristupiti iz svih imenovanih prostora koji su u ovoj datoteci, bez klasifikacije. System imenovani prostor je koreni imenovani prostor za aplikacije .NET okruženja i sadrži sve osnovne funkcionalnosti neophodne za konzolne aplikacije.

Zatim je deklarisan i imenovani prostor za samu aplikaciju, *Pozdrav Svete*.

Sažetak

U ovom predavanju prešli smo priličan deo terena koji se bavi pravljenjem korisnih (ako ne i osnovnih) C# aplikacija. Upoznali smo se sa osnovama C# sintakse i analizirali osnove koda konzolnih aplikacija koje VS generiše za nas, kada se pravi konzolni aplikacioni projekat.
Najveći deo ovog predavanja, bavi se korišćenjem promenljivih. Videli smo šta su to promenljive, kako se prave, kako im se dodeljuju vrednosti i kako se manipuliše tim vrednostima. Usput, upoznali smo se i sa osnovama interakcije sa korisnikom, tako što smo videli kako ispisati tekst u konzolnoj aplikaciji i kako pročitati ono što nam korisnik unese. Ovo je uključilo samo osnovne tipove konverzije.

Takođe smo naučili kako formirati izraze pridruživanjem operatora i operanda, i kako se ovi izvršavaju i kojim redosledom.

Na kraju, upoznali smo se sa pojmom imenovanih prostora, koji će nam kako napredujemo kroz knjigu postajati sve važniji. Pošto vam je ova tema predstavljena prilično skraćeno, stvorena je osnova za dalje diskusije.

Sve što smo do sada programirali izvršava se red po red.

Vežbe

1. Kako pozvati ime **veliki** iz koda unutar imenovanog prostora **cudan** ?

```
namespace cudan
{
   //kod u imenovanom prostoru cudan
}

namespace super
{
   namespace velikani
   {
     // definisano ime veliki
   }

}
```

2. Koje od navedenih imena promenljivih nije ispravno:

   a) mojaPromenljivaJeDobra
   b) 99Flake
   c) _floor
   d) time2GetJiggyWidIt
   e) tf.zr.ac.yu

Kontrola toka programa

Sav kod u jeziku C# koji smo do sada videli ima nešto zajedničko. U svakom od primera kod se izvršavao red po red, od gore ka dole, ne propuštajući ništa. Kada bi se svaka aplikacija izvršavala ovako, bili bismo vrlo ograničeni u onome što se da uraditi.
U ovom predavanju videćemo dve metode kontrole toka programa, odnosno redosleda izvršenja redova u C# kodu. Te dve metode su sledeće:

- Grananje - gde izvršavamo kod uslovno, u zavisnosti od rezultata nekog poređenja, na primer. „Ovaj kod izvrši samo ako je myVal manje od 10."
- Petlje - ponavljanje izvršenja istih iskaza (određeni broj puta, dok se ne dostigne uslov koji je unapred zadat).

Obe tehnike uključuju korišćenje Bulove logike. U predavanju o prikazu tipova videli smo tip bool, ali nismo mnogo radili sa njim. Krenućemo od toga šta se podrazumeva pod Bulovom logikom, tako da bismo je mogli upotrebiti u toku kontrole programa.

Bulova logika

Tip bool, može sadržati samo dve vrednosti: true ili false. Ovaj tip se često koristi da bi se zabeležio rezultat neke operacije, tako da možemo reagovati na osnovu tog rezultata. Uglavnom se tipovi bool koriste da uskladište rezultat nekog poređenja.

Istorijski, vredelo bi sećati se (i poštovati) engleskog matematičara Džordža Bula, čiji je rad sredinom devetnaestog veka omogućio osnovu Bulove logike.

Kao primer, razmotrite situaciju u kojoj bismo želeli da izvršimo kod na osnovu toga da li je promenljiva myVal manja od 10. Da bismo ovo uradili, moramo proveriti da li je iskaz „myVal je manje od 10" istinit ili neistinit. Tačnije, moramo znati logičku vrednost rezultata tog poređenja.
Logičko poređenje zahteva upotrebu logičkog operatora poređenja (koji se još zove i relacioni operator), koji su prikazanih u donjoj tabeli. U svim ovim slučajevima, var1 je promenljiva tipa bool, dok tipovi promenljivih var2 i var3 mogu biti različiti.

Tabela 5.1. Operatori dodeljivanja

Operator	Kategorija	Primer izraza	Rezultat
==	binarni	var1=var2== var3;	var1 je dodeljena vrednost true ako je var2 jednaka var3, u suprotnom false.
!=	binarni	var1=var2!= var3;	var1 je dodeljena vrednost true ako je var2 nije jednaka var3, u suprotnom false.
<	binarni	var1=var2< var3;	var1 je dodeljena vrednost true ako je var2 manja od var3, u suprotnom false.
>	binarni	var1=var2>var3;	var1 je dodeljena vrednost true ako je var2 veća od var3, u suprotnom false.
<=	binarni	var1=var2<= var3;	var1 je dodeljena vrednost true ako je var2 manja ili jednaka var3, u suprotnom false.
>=	binarni	var1=var2>= var3;	var1 je dodeljena vrednost true ako je var2 veća ili jednaka var3, u suprotnom false.

Možemo koristiti ove operatore na brojnim vrednostima u kodu kao što je:

```
bool isLessThan1O;                                          .      ...
isLessThan1O = myVal < 10;
```

U ovom kodu biće dodeljena vrednost true promenljivoj **isLessThan1O**, ako je vrednost promenljive myVal manja od 10, u suprotnom biće dodeljena vrednost false. Ove operatore poređenja možemo koristiti i na drugim tipovima, kao što su stringovi:

```
bool  isPetar;
isPetar =myString  ==  "Petar";
```

Ovde će **isPetar** biti istinita samo ako myString u sebi ima string "**Petar**". Takođe se možemo fokusirati i na logičke vrednosti:

```
bool isTrue;
isTrue = myBool == true;
```

Ipak, ovde smo ograničeni na korišćenje samo operatora == i ! =.

Upamtite da će doći do greške u kodu ako slučajno pretpostavite da va11>va12 mora biti true zato što je va11<va12 false. Ako je va11 == va12 onda su oba ova iskaza false.

Postoje i drugi logički operatori koji su namenjeni isključivo radu sa logičkim vrednostima:

Tabela 5.2. Operatori dodeljivanja

Operator	Kategorija	Primer izraza	Rezultat
!	unarni	var1=!var2;	var 1 je dodeljena vrednost true ako je var2 false, ili vrednost false ako je var2 true (logičko NE).
&	binarni	var1=var2& var3;	var1 je dodeljena vrednost true ako var2 i var3 imaju vrednost true u suprotnom false (logičko I).
|	binarni	var1=var21 var3;	var1 je dodeljena vrednost true ako var2 ima vrednost true ili var3 ima vrednost true ili obe promenljive imaju vrednost true, u suprotnom dobija vrednost false (logičko ILI).
^	binarni	var1=var2 var3;	var1 je dodeljena vrednost true ako var2 ima vrednost true ili var3 ima vrednost true, ali ne i ako obe promenljive imaju vrednost true, u suprotnom false (Logičko ekskluzivno ILI).

Tako poslednji primer koda može biti napisan i kao:

```
bool  isTrue;
isTrue = myBool  &  true;
```

Operatori & i I imaju dva slična operatora:

Tabela 5.3. Operatori dodeljivanja

Operator	Kategorija	Primer izraza	Rezultat
&&	binarni	var1=var2&&var3;	var1 je dodeljena vrednost true ako var2 i var3 imaju vrednost true, u suprotnom false (logičko I)
||	binarni	var1=var2||var3;	var1 je dodeljena vrednost true ako var2 ili var3 (ili obe promenljive) imaju vrednost true, u suprotnom var1 dobija vrednost false (logičko ILI)

Njihov rezultat je potpuno isti kao & i |, ali postoji bitna razlika u načinu na koji se dobija njihov rezultat, što može dati bolje performanse programa. Oba operatora prvo ispituju vrednost prvog operanda (var2 u tabeli gore), pa na osnovu vrednosti ovog operanda možda neće biti potrebno da se ispita drugi operand (var 3 gore).

Ako je vrednost prvog operanda operatora && false, onda nema potrebe ispitivati vrednost drugog, zato što će rezultat, bez obzira na drugi operand biti false. Slično tome operator | | vratiće vrednost true, ako je prvi operand true, bez obzira na vrednost drugog operanda.
Ovo nije slučaj sa operatorima & i | koje smo videli pre toga. Kod njih će oba operanda uvek biti izračunata.

Zbog ovog uslovnog izračunavanja operanada, primetićemo malo poboljšanje performansi ako koristimo operatore & & i | | umesto & i |. Ovo će biti posebno očigledno u aplikacijama u kojima se često upotrebljavaju ovi operatori. Neko opšte pravilo je da uvek treba koristiti operatore & & i I I gde god je to moguće.

Iskaz goto

Jezik C# dozvoljava da se određeni delovi koda obeleže i da se na ta mesta direktno skače pomoći j iskaza goto, što nam koristi, ali i stvara probleme. Glavna korist je u tome što možemo na jednostavan način kontrolisati šta se u kodu izvršava i kada. Glavni problem leži u preteranoj upotrebi, jer se na taj način stvara nerazumljiv špageti kod.

Pogledajmo kako da potvrdimo ovo što smo naveli Iskaz goto se koristi na sledeći način:

goto <imeOznake>;

Oznake se definišu na sledeći način:

<imeOznake>:

Na primer razmotrimo sledeće:

```
int myInteger =  5;
goto myLabel;
myInteger +=10;
myLabel:
Console.WriteLine("myInteger =  {0}",  myInteger);
```

Program se izvršava na sledeći način:

- Promenljiva mylnteger je deklarisana kao tip int i dodeljena joj je vrednost 5
- Iskaz **goto** prekida normalan tok programa i prebacuje kontrolu na red obeležen sa myLabel:
- Vrednost mylnteger se ispisuje na konzoli.

Naznačeni deo koda ispod, nikada se ne izvršava:

```
int myInteger =  5;
goto myLabel;
myInteger +=  10;
myLabel:
Console.WriteLine("myInteger  =  {0}",  myInteger);
```

Ako isprobate ovo u nekoj aplikaciji, videćete da je, kad pokušate da kompajlirate, u listi zadataka navedeno upozorenje "Unrechable code detected", zajedno sa brojem reda.
Iskazi goto imaju svoju upotrebu, ali mogu prilično zakomplikovati stvari. Kao primer špageti koda proisteklog iz iskaza goto, pogledajte sledeće:

```
start:
int myInteger = 5;
goto addVal;
writeResult:
Console. WriteLine ("myInteger =  (0)",  myInteger);
goto start;
addVal:
myInteger +=10;
goto writeResult;
```

Ovaj kod je potpuno ispravan, ali je nečitljiv! Ako želite da ovo sami isprobate, pokušajte da vidite šta ovaj kod radi samo gledanjem u njega.
Vratićemo se ovom iskazu i svim njegovim implikacijama malo kasnije.

Grananje

Grananje je čin kontrole toga koji red koda treba da se izvrši sledeći. Red na koji se skače zavisi od neke vrste uslovnog iskaza. Taj uslovni iskaz biće zasnovan na poređenju između test vrednosti i jedne ili vise mogućih vrednosti sa korišćenjem Bulove logike.
U ovom delu pogledaćemo tri tehnike grananja dozvoljenih u jeziku C#:

- ternarni operator
- iskaz **if**
- iskaz **switch**

Ternarni operator

Najjednostavniji način da se izvede poređenje jeste pomoću ternarnog (ili uslovnog) operator!. Već smo videli unarne operatore koji rade sa jednim operandom kao i binarne koji rade sa dva operanda, tako da ne čudi što ternarni koristi tri operanda. Sintaksa je sledeća:

<test> ? <resultIfTrue> : <resultIfFalse>

Ovde se ispituje **<test>** da bi se dobila logička vrednost, pa je rezultat operator **<resultIfTrue>** ili **<resultIfFalse>** u zavisnosti od te vrednosti.

Ovo možemo koristiti u sledećem:

```
string resultString =  (myInteger <  10)          ?  "Manje od  10"
                                                  : Veće ili  jednako  10";
```

Ovde je rezultat ternarnog operatora jedan od dva stringa, od kojih oba mogu biti dodeljeni promenljivoj **resultString**. Izbor se vrši na osnovu poređenja vrednosti promenljivi **myInteger** sa 10, pa ukoliko je **myInteger** manje od **10**, prvi string se smešta i **resultString**, a ukoliko je veće ili jednako **10**, **resultString** se upisuje u drugi string. Na primer, ako je **myInteger 4**, onda je **resultString** string "**Manje od 10**".
Ovaj operator je dobar za male zadatke nalik ovom, ali nije odgovarajući za velike količine kodi bazirane na poređenju. Mnogo bolji način da se to uradi jeste pomoću iskaza **if**.

Iskaz if

Iskaz if je mnogo bolji i raznovrsniji način za donošenje odluka. Za razliku od iskaza ?:, iskaz iff nema rezultat (pa ga zato ne koristimo prilikom dodele); umesto toga, ovaj iskaz koristimo dii uslovno izvršimo druge iskaze.

Najjednostavnije korišćenje iskaza if je sledeće:

```
If (<test>):
    <kod se izvršava ako je test istinit >;
```

Ovde se **<test>** izračunava (mora biti logička vrednost da bi se kod kompajlirao), a iskaz ispod se izvršava ako uslov **<test>** ima vrednost **true**. Nakon izvršenja tog koda, ili ukoliko <test> imao rezultat false, program se nastavlja od sledećeg reda koda.

Možemo navesti i dodatni kod koristeći iskaz **else** u kombinaciji sa iskazom if. Ovaj iskaz se izvršava ako **<test>** ima vrednost false:

```
if (<test>)
    <kod se izvršava ako je <test> true>;
else
    <kod se izvršava ako je <test> false>;
```

Oba dela ovog koda mogu se prostirati u više redova ograničenih vitičastim zagradama:

```
if (<test>)
{
    <kod se izvršava ako je <test> true>;
}
else
{
    <kod se  izvršava ako je  <test>  false>;
}
```

Kao kratak primer, prepravimo kod u kome smo koristili ternarni operator:

```
string resultString = (myInteger < 10) ? "Manje od 10" :
                "Veće ili jednako 10";
```

Pošto rezultat iskaza if ne možemo dodeliti promenljivoj, vrednost promenljivoj se mora dodeliti u posebnom koraku:

```
string resultString;
if  (myInteger < 10)
    resultString = "Manje od 10"
else
    resultString  =  "Veće ili jednako 10";
```

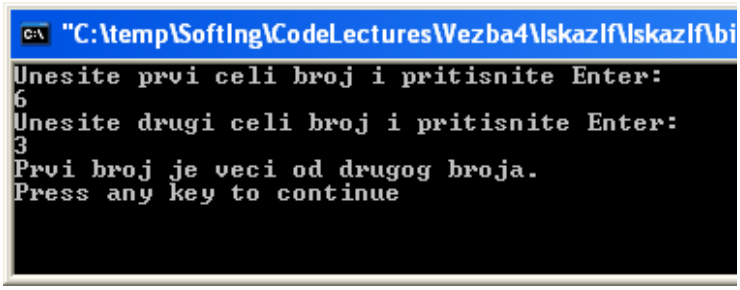
Kod kao ovaj, iako je složeniji, mnogo je lakši za čitanje i razumljiviji je od ekvivalentnog u ternarnom obliku, a dozvoljava i veću fleksibilnost.
Uradimo vežbu.

Vežba br. 4.

Napravite novi projekat konzolne aplikacije birajući File | New | Project... iz menija: Izaberite Visual C# Projects direktorijum unutar prozora Project Types: i tip projekta Console Application u okviru prozora Templates: (za ovo ćete morati malo da pomerite prozor na dole). U okviru za tekst Location: promenite putanju u **C:\Temp\SoftIng\LecturesCode\Vezba4** (ovaj će direktorijum biti automatski napravljen ukoliko već ne postoji), i ostavite podrazumevani tekst u okviru za tekst Name: **IskazIf**
Dodajte sledeći kod u Class1.cs:

```
13      [STAThread]
14      static void Main(string[] args)
15      {
16        string komparacija;
17        Console.WriteLine("Unesite prvi celi broj i pritisnite Enter:");
18        double prom1 = Convert.ToDouble(Console.ReadLine());
19        Console.WriteLine("Unesite drugi celi broj i pritisnite Enter:");
20        double prom2 = Convert.ToDouble(Console.ReadLine());
21        if (prom1<prom2)
22          komparacija = "manji";
23        else
24        {
25          if (prom1 ==prom2)
26            komparacija = "jednak";
27          else
28            komparacija = "veći";
29        }
30        Console.WriteLine("Prvi broj je {0} od drugog broja.", komparacija);
31      }
32    }
33  }
```

Pokrenite program i unesite dva broja kad zatraži:



Kako to radi

Prvi deo koda vam je poznat i on obezbeđuje unos vrednosti tipa double od korisnika:

```
string komparacija;
Console.WriteLine("Unesite prvi celi broj i pritisnite Enter:");
double prom1 = Convert.ToDouble(Console.ReadLine());
Console.WriteLine("Unesite drugi celi broj i pritisnite Enter:");
double prom2 = Convert.ToDouble(Console.ReadLine());
```

Zatim dodeljujemo string promenljivoj komparacija tipa string, na osnovu vrednosti koje sadrže promenljive prom1 i prom2. Prvo proveravamo da li je prom1 manje od prom2:

```
if (prom1<prom2)
  komparacija = "manji";
```

Ukoliko to nije slučaj, prom1 je ili veće ili jednako prom2. Unutar dela **else** prvog poređenja moramo da ugnezdimo drugo poređenje.:

```
else
{
  if (prom1 ==prom2)
    komparacija = "jednak";
```

Do **else** dela drugog poređenja doći će se samo ako je prom1 veće od prom2:

```
    else
      komparacija = "veći";
}
```

I na kraju ispisujemo vrednost poređenja na konzolu:

```
Console.WriteLine("Prvi broj je {0} od drugog broja.", komparacija);
```

Ugnježdavanje koje smo koristili jeste samo jedan od načina. Isto tako smo mogli da napišemo da ispitijumo svaki put promenljive

Manjkavost ove metode jeste u tome što se izvode tri poređenja, bez obzira na vrednosti prom1 i prom2. Prvom metodom izvodimo samo jedno poređenje ako prom1<prom2 ima vrednost true i dva poređenja u suprotnom (takođe izvodimo poređenje prom1 == prom2), što zajedno čini manji broj redova koje se izvršavaju. Ovde će razlike u performansama biti minimalne, ali to neće biti slučaj i u aplikacijama kojima je jako bitna brzina izvršavanja.

Proveravanje uslova korišćenjem iskaza if

U prethodnom primeru proveravali smo tri uslova koji uključuju vrednost prom1. Ovo je pokrilo sve moguće vrednosti promenljive. U drugim slučajevima možda ćemo želeti da proverimo specifične vrednosti, recimo ako je prom1 jednako 1, 2, 3 ili 4, ili slično. Ako koristimo kod nalik onome od malopre, to može prouzrokovati vrlo neprijatno ugnežden kod, na primer:

```
if (prom1  ==  1)
{
  // uradi nešto
}
else
{
  if (prom1 == 2)
  {
    // uradi nešto drugo
  }
  else
  {
    if (prom == 3)
    {
      // uradi nešto drugo
    }
    else
    {
      // uradi nešto drugo
    }
  }
}
```

Upamtite da je česta greška pisati kod kao što je treći uslov u obliku if (prom1 ==3 || == 4). Ovde, redosledu izvršenja, operator == se izvršava prvi, ostavljajući operatoru | | da radi sa logičkim i numeričkim operandom. Ovo će prouzrokovati grešku.

U ovakvim situacijama vredi koristiti malo izmenjenu šemu uvlačenja teksta i spajanje bloka koda za else blokove (odnosno jednog reda posle else bloka koda, umesto novog bloka). Kada se ovo uradi, dobije se struktura koja uključuje else if iskaz:

```
if   (prom1 ==  1)
{
  // uradi nešto
}
```

Iskaz else if predstavlja u stvari, dva odvojena iskaza, dok je kod funkcionalno identičan onome pre njega. Pa ipak, ovaj kod je mnogo lakši za čitanje.
Kada pravite višestruka poređenja kao što je ovo, vredi razmisliti o iskazu switch kao alternativnoj strukturi grananja.

Iskaz switch

Iskaz switch je vrlo sličan iskazu if po načinu uslovnog izvršavanja koda na osnovu neke vrednosti dobijene iz testa. Međutim, switch nam dozvoljava da ispitamo vise različitih vrednosti test promenljive odjednom, umesto samo jednog uslova. Ovaj test je ograničen na diskretne vrednosti, umesto na članove kao što je " veće od X", pa je i njegova upotreba malo drugačija, ali i dalje može biti moćna tehnika.
Osnovna struktura switch iskaza je sledeća:

```
switch (<testProm>)
{
  case <constIzraz11>:
      <kod koji se izvršava ako je <testProm> == <constIzraz11> >
      break;
  case <constIzraz12>:
      <kod koji se izvršava ako je <testProm> == <constIzraz12> >
      break;
      ....
      ....
  case <constIzraz1N>:
      <kod koji se izvršava ako je <testProm> == <constIzraz1N> >
      break;
  default::
      <kod koji se izvršava ako je <testProm> != <constIzrazi> >
      break;
}
```

Vrednost u promenljivoj <**testProm**> poredi se sa svakom od vrednosti <**constIzrazX**> (navedenim u iskazu case) i ako postoji poklapanje, onda se izvršava kod predviđen za to. Ukoliko ne postoji poklapanje ni sa jednom vrednošću, onda se izvršava kod u delu **default**, ako takav blok postoji. Na kraju koda svake sekcije imamo dodatnu komandu **break**. Nelegalno je za tok izvršenja programa da prede u sledeći case iskaz, ako se obradio case blok pre njega.

Upamtite da se ova oblast u jeziku C#, razlikuje od jezika C++ gde je dozvoljeno izvršenje jednog case iskaza iza drugog.

Iskaz break ovde jednostavno obustavlja iskaz switch i prelazi na sledeći iskaz prate strukturu programa.
U okviru jezika C# postoje alternativne metode sprečavanja toka izvršenja jednog case iskaza iza drugog. Možemo koristiti iskaz return, koji obustavlja izvršenje trenutne funkcije a ne samo switch strukture, ili goto iskaz. Iskaz goto (kao što je ranije navedeno) ovde radi, zato što iskaz case u stvari definiše oznake u C# kodu. Na primer:

```
switch (<testProm>)
{
  case <constIzraz11>:
      <kod koji se izvršava ako je <testProm> == <constIzraz11> >
      break;
  goto case <constIzraz12>:
      <constIzraz12>:
      <kod koji se izvršava ako je <testProm> == <constIzraz12> >
      break;
      ....
      ....
```

Ovde postoji izuzetak od pravila da izvršavanje jednog case iskaza ne može slobodno da prede u sledeći. Ako postavimo vise case iskaza zajedno pre jednog bloka koda, mi u stvari proveravamo višestruke uslove odmah. Ako je bilo koji od uslova ispunjen, kod se izvršava.

Na primer:

```
switch (<testProm>)
{
  case <constIzraz11>:
  case <constIzraz12>:
      <kod koji se izvršava ako je <testProm> == <constIzraz11> > ili
                                          <testProm> == <constIzraz12> >
      break;
      ....
      ....
```

Upamtite da se ovi uslovi odnose takođe i na iskaz default. Ne postoji nikakvo pravilo po kome ovaj iskaz mora biti zadnji u listi poređenja, pa ga možemo postaviti u grupu sa iskazima **case**, ako to želimo. Dodavanje tačke prekida pomoću **break**, **goto** ili **return** osigurava pravilno izvršenje putanje kroz strukturu u svim iskazima case.

Svaki od poređenja <**constIzrazX**> mora biti konstantna vrednost. Jedan od načina da se ovo postigne jeste da obezbedimo vrednost literala, na primer:

```
switch (<mojInteger>)
{
  case 1:
      <kod koji se izvršava ako je mojInteger == 1> >
      break;
  case -1:
      <kod koji se izvršava ako je mojInteger == -1> >
      break;
  default :
      <kod koji se izvršava ako je mojInteger != poređenja>
      break;
}
```

Još jedan od načina jeste i da koristite **konstantne promenljive**. Konstantna promenljiva je nalik bilo kojoj drugoj promenljivoj, osim što se razlikuje u ključnoj stvari - vrednost koju sadrži nikad se ne menja. Jednom kad dodelimo vrednost konstantnoj promenljivoj, ta vrednost ostaje do kraja izvršenja programa. Konstantne promenljive nam u ovom slučaju pomažu, zato što je često lakše čitati kod gde su prave vrednosti koje se upoređuju skrivene od nas u vreme poređenja.

Konstantne promenljive se deklarišu korišćenjem ključne reči const, kao dodatak tipu promenljive i pri torn im moraju biti dodeljene vrednosti, na primer:

```
const  int  intTwo =  2;
```

Ovaj kod je potpuno ispravan, ali ako pokušamo:

```
const  int intTwo;
intTwo =  2;
```

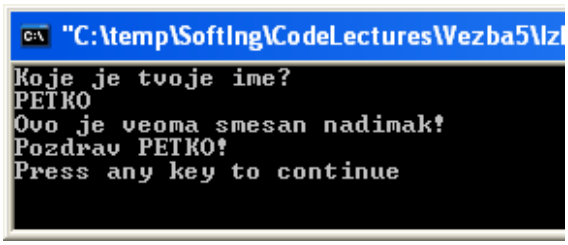
pojaviće se greška pri kompajliranju. Ovo će se dogoditi i ako pokušamo da promenimo vrednost I konstantne promenljive, na bilo koji način posle prve dodele vrednosti.
Pogledajmo primer iskaza switch koji koristi konstantne promenljive.

Vežba br. 5.

Napravite novi projekat konzolne aplikacije birajući File | New | Project... iz menija:
Izaberite Visual C# Projects direktorijum unutar prozora Project Types: i tip projekta Console Application u okviru prozora Templates: (za ovo ćete morati malo da pomerite prozor na dole).U okviru za tekst Location,promenite putanju u **C:\Temp\SoftIng\LecturesCode\Vezba5** (ovaj će direktorijum biti automatski napravljen ukoliko već ne postoji), i ostavite podrazumevani tekst u okviru za tekst Name: **IskazSwitch.**
Dodajte sledeći kod u Class1.cs:

```
14      static void Main(string[] args)
15      {
16        const string mojeIme = "petar";
17        const string mojePrezime = "petrović";
18        const string mojNadimak = "petko";
19        string ime;
20        Console.WriteLine("Koje je tvoje ime?");
21        ime = Console.ReadLine();
22        switch (ime.ToLower())
23        {
24          case mojeIme:
25            Console.WriteLine("Ti imas isto ime kao moje!");
26            break;
27          case mojePrezime:
28            Console.WriteLine("Ti imas isto Prezime kao moje!");
29            break;
30          case mojNadimak:
31            Console.WriteLine("Ovo je veoma smešan nadimak!");
32            break;
33        }
34        Console.WriteLine("Pozdrav {0}!", ime);
35      }
36    }
37  }
```

Pokrenite program i unesite ime:



Kako to radi

Naš kod postavlja tri konstante tipa string, prihvata promenljivu tipa string od korisnika, a zatim ispisuje tekst na konzolu, zasnovan na tome šta je uneto. U ovom slučaju promenljive tipa string su imena.
Kada uporedimo uneseno ime (u promenljivoj **ime**) sa konstantnom vrednošću, prvo ga prebacujemo u mala slova pomoću metoda **ime.ToLower()** . To je standardna komanda koja će raditi sa svim promenljivama tipa string i priskače nam u pomoć kada nismo sigurni šta je korisnik uneo. Koriste ovu tehniku stringovi "Petar", "peTAR", "petar" itd., poklopi će se sa test stringom "petar".

Iskaz switch sam po sebi pokušava da pronađe poklapanje unetog stringa sa konstantnom vrednošću koja je definisana i ispisuje ličnu poruku da, ako uspe, pozdravi korisnika. Ako ne pronađemo poklapanje, samo pozdravljamo korisnika.

Iskaz switch nema ograničenja na broj iskaza case koje sadrži, tako da možete proširiti ovaj kod za svako ime koje vam padne na pamet, ako želite, samo što će potrajati.

Petlje

Petlje predstavljaju ponavljanje izvršenja iskaza. Ova tehnika može biti od velike pomoći, jer omogućava ponavljanje operacije koliko god hoćemo puta (na hiljade, ili čak milione puta), bez potrebe da svaki put pišemo istu.

Kao primer, razmotrimo sledeći kod za izračunavanje količine novca na bankovnom računu posle deset godina, pretpostavljajući da je kamata plaćena svake godine i da nema drugih priliva ili odliva sa računa:

```
double balance =  1000;
double interestRate =  1,05;  // 5% kamate godišnje
balance *= interestRate;
balance *= interestRate;
balance *= interestRate;
balance *= interestRate;
balance *= interestRate;
balance *= interestRate;
balance *= interestRate;
balance *= interestRate;
balance *= interestRate;
balance *= interestRate;
```

Pisanje istog koda 10 puta deluje kao razbacivanje vremena, a i šta ako moramo da promenimo sa 10 na neki drugi broj godina ? Morali bismo da ručno iskopiramo red po red traženi broj puta, što bi bilo naporno.
Umesto toga možemo da koristimo petlju koja izvršava instrukcije koje mi želimo određen broj puta.

Još jedan važan tip petlji jeste onaj gde ostajemo u petlji dok se neki uslov ne ispuni. Ove petlje su malo jednostavnije (lako ništa manje korisne), pa ćemo početi sa njima.

Petlje do

Petlje do rade na sledeći način: kod koji je obeležen unutar petlje se izvršava; zatim se izvodi logički test; pa se kod izvršava ponovo ako je rezultat tog testa true i tako dalje. Iz petlje se izlazi kada rezultat testa postane false.

Struktura do petlje izgledao ovako:

```
do
{
    <kod koji se ponavlja u petlji>
}   while (<test>);
```

gde je **<test>** logička vrednost.

Tačka-zarez posle while iskaza je neophodna. Uobičajena greška je da se ona zaboravi.

Ovo možemo koristiti da bismo recimo, napisali brojeve od 1 do 10 u koloni:

```
int i=1;
do
{
    Cosole.WriteLine(“{0}“, i++);
}   while (i<=10);
```

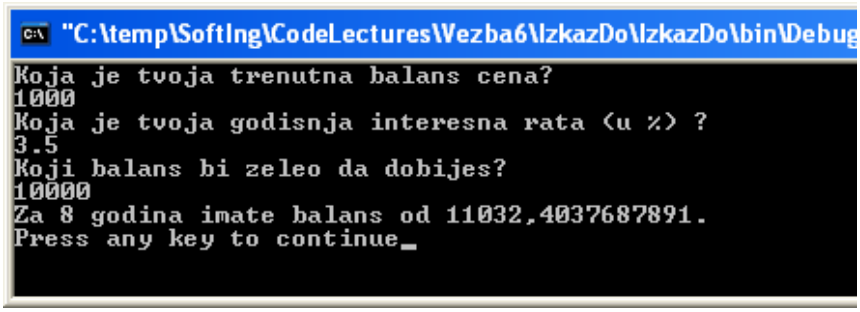
Ovde koristimo sufiks verziju operatora ++ da bismo povećali vrednost promenljive i pošto se ispiše na ekranu, tako da moramo proveriti da li je i<=10, da bismo uključili i 10 u brojeve koji se ispisuju na konzolu.
Iskoristimo ovo da malo modifikujemo kod koji smo u uvodu ovog dela knjige koristili da Izračunamo stanje na računu posle 10 godina. Ovde ćemo koristiti petlju da bismo izračunali koliko je godina potrebno da dođemo do određene sume novca, na osnovu početne sume i kamatne stope.

Vežba br. 6.

Napravite novi projekat konzolne aplikacije birajući File | New | Project... iz menija:
Izaberite Visual C# Projects direktorijum unutar prozora Project Types: i tip projekta Console Application u okviru prozora Templates: (za ovo ćete morati malo da pomerite prozor na dole).U okviru za tekst Location,promenite putanju u **C:\Temp\SoftIng\LecturesCode\Vezba6** (ovaj će direktorijum biti automatski napravljen ukoliko već ne postoji), i ostavite podrazumevani tekst u okviru za tekst Name: **IskazDo.**
Dodajte sledeći kod u Class1.cs:

```
14      static void Main(string[] args)
15      {
16        double balans, intRata, ciljniBalans;
17        Console.WriteLine("Koja je tvoja trenutna balans cena?");
18        balans=Convert.ToDouble(Console.ReadLine());
19        Console.WriteLine("Koja je tvoja godišnja interesna rata (u %) ?");
20        intRata=1+Convert.ToDouble(Console.ReadLine())/100;
21        Console.WriteLine("Koji balans bi želeo da dobiješ?");
22        ciljniBalans=Convert.ToDouble(Console.ReadLine());
23
24        int brojGodina=0;
25        do
26        {
27          balans *= intRata;
28          ++brojGodina;
29        }
30          while (balans < ciljniBalans);
31          Console.WriteLine("Za {0} godin{1} imate balans od {2}.",
32            brojGodina, brojGodina==1 ? "u": "a", balans);
33      }
34    }
35  }
```

3. Pokrenite aplikaciju i unesite neke vrednosti:



Kako to radi

Ovaj kod prosto ponavlja jednostavnu godišnju kalkulaciju stanja, sa fiksnom kamatnom stopom, onoliko puta koliko je potrebno da bi se zadovoljio zadati uslov koji okončava petlju. Brojimo koliko godina je potrebno da se izračunava kamata, tako što povećavamo brojačku promenljivu za svaki novi ciklus u petlji:

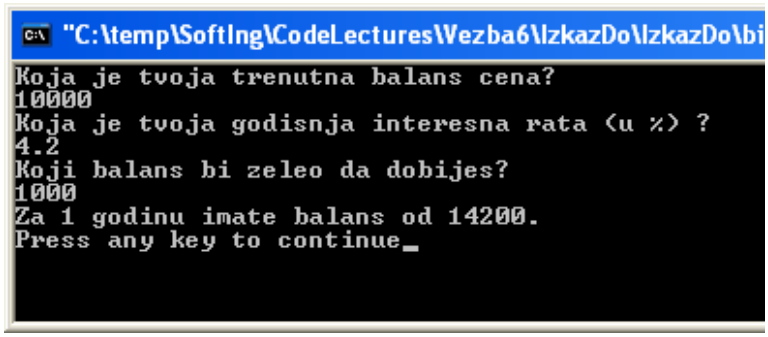
```
int brojGodina=0;
do
{
  balans *= intRata;
  ++brojGodina;
}
  while (balans < ciljniBalans);
```

Ovu brojač promenljivu možemo zatim koristiti u izlaznom rezultatu:

```
Console.WriteLine("Za {0} godin{1} imate balans od {2}.",
  brojGodina, brojGodina==1 ? "u": "a", balans);
```

Upamtite daje ovo možda najčešća upotreba ternarnog operatora?: da bise uslovno formatirao tekst sa minimumom koda. Ovde stavljamo “a” posle „godin“ ako brojGodina nije jednaka 1.

Na žalost, ovaj kod nije savršen. Razmotrite situaciju u kojoj je traženo stanje manje od trenutnog stanja. U ovom slučaju izlazni podaci će biti:



do petlje uvek izvršavaju bar jedan ciklus. Ponekad, kao u ovoj situaciji, to nije idealno. Naravno, možemo dodati iskaz if:

```
14      static void Main(string[] args)
15      {
16        double balans, intRata, ciljniBalans;
17        Console.WriteLine("Koja je tvoja trenutna balans cena?");
18        balans=Convert.ToDouble(Console.ReadLine());
19        Console.WriteLine("Koja je tvoja godišnja interesna rata (u %) ?");
20        intRata=1+Convert.ToDouble(Console.ReadLine())/100;
21        Console.WriteLine("Koji balans bi želeo da dobiješ?");
22        ciljniBalans=Convert.ToDouble(Console.ReadLine());
23
24        int brojGodina=0;
25        if (balans<ciljniBalans)
26        {
27          do
28          {
29            balans *= intRata;
30            ++brojGodina;
31          }
32          while (balans < ciljniBalans);
33          Console.WriteLine("Za {0} godin{1} imate balans od {2}.",
34            brojGodina, brojGodina==1 ? "u": "a", balans);
```

Ali ovo izgleda kao da bespotrebno komplikujemo kod. Mnogo bolje rešenje je koristiti petlju while.

Petlje while

Petlje while su dosta slične do petljama sa jednom bitnom razlikom: logički test se u petljama while izvršava na početku, a ne na kraju ciklusa. Ukoliko je rezultat testa false, ciklus petlje se nikad ne izvrši. Umesto toga izvršenje programa odmah skače na red koda koji se nalazi iza petlje.
Petlje while se definišu na sledeći način:

```
while (<test>)
{
  <kod unutar petlje>
}
```

Mogu se koristiti skoro na isti način kao i petlja do, na primer:

```
int i=1;
while (i<=10)
{
Console.WriteLine(" (0)", i++);
}
```

Ovaj kod nam daje isti rezultat kao do petlja koju smo ranije videli - ispisuje brojeve od 1 do 10 u koloni. Izmenimo poslednji primer tako da iskoristimo petlju while.

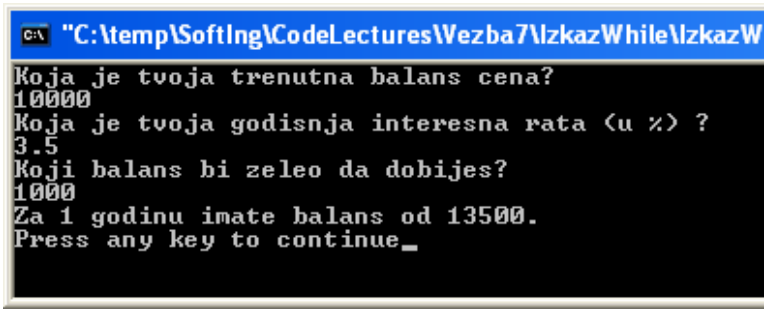
Vežba br. 7.

Napravite novi projekat konzolne aplikacije birajući File | New | Project... iz menija: Izaberite Visual C# Projects direktorijum unutar prozora Project Types: i tip projekta Console Application u okviru prozora Templates: (za ovo ćete morati malo da pomerite prozor na dole).U okviru za tekst Location,promenite putanju u **C:\Temp\SoftIng\LecturesCode\Vezba7** (ovaj će direktorijum biti automatski napravljen ukoliko već ne postoji), i ostavite podrazumevani tekst u okviru za tekst Name: **IskazWhile.**
Dodajte sledeći kod u Class1.cs:

Prepravite kod na sledeći način tako što iskoristite kod iz prethodnog primera, samo obrišite iskaz while na kraju originalne petlje.

```
14      static void Main(string[] args)
15      {
16        double balans, intRata, ciljniBalans;
17        Console.WriteLine("Koja je tvoja trenutna balans cena?");
18        balans=Convert.ToDouble(Console.ReadLine());
19        Console.WriteLine("Koja je tvoja godišnja interesna rata (u %) ?");
20        intRata=1+Convert.ToDouble(Console.ReadLine())/100.0;
21        Console.WriteLine("Koji balans bi želeo da dobiješ?");
22        ciljniBalans=Convert.ToDouble(Console.ReadLine());
23
24        int brojGodina=0;
25        //while (balans < ciljniBalans);
26          {
27            balans *= intRata;
28            ++brojGodina;
29          }
30        Console.WriteLine("Za {0} godin{1} imate balans od {2}.",
31            brojGodina, brojGodina==1 ? "u": "a", balans);
32      }
33    }
34  }
35
```

Pokrenite ponovo aplikaciju, ali ovaj put unesite traženu vrednost stanja koja je manja od početnog stanja:



Kako to radi

Ova jednostavna promena iz petlje do u petlju while, rešava problem u zadnjem primeru. Tako što smo premestili logički test sa kraja petlje na početak, obezbedili smo se za situacije

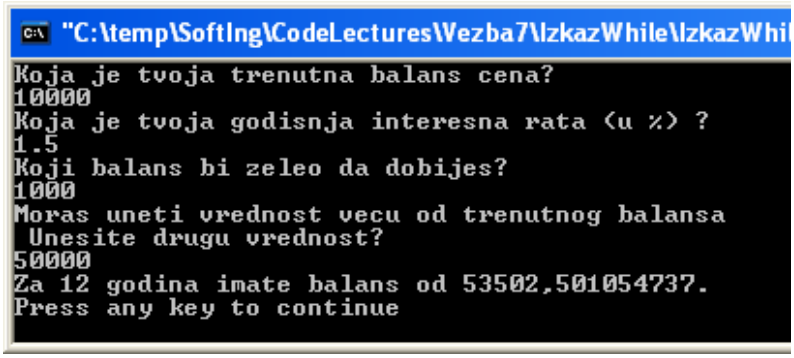
kada nije potrebno izvršavati petlju, tako da možemo odmah skočiti na rezultat.

Postoje naravno i alternative za ovakvu situaciju. Na primer, možemo proveriti korisnikov unos da bismo se osigurali da je traženo stanje veće od početnog stanja.

U ovakvim situacijama možemo postaviti unos korisnika u petlju na sledeći način:

```
14      static void Main(string[] args)
15      {
16        double balans, intRata, ciljniBalans;
17        Console.WriteLine("Koja je tvoja trenutna balans cena?");
18        balans=Convert.ToDouble(Console.ReadLine());
19        Console.WriteLine("Koja je tvoja godišnja interesna rata (u %) ?");
20        intRata=1+Convert.ToDouble(Console.ReadLine())/100.0;
21        Console.WriteLine("Koji balans bi želeo da dobiješ?");
22        do
23        {
24          ciljniBalans=Convert.ToDouble(Console.ReadLine());
25          if (ciljniBalans<=balans)
26            Console.WriteLine("Moraš uneti vrednost vecu " +
27                    "od trenutnog balansa\n Unesite drugu vrednost?");
28        }
29        while (ciljniBalans<=balans);
30        int brojGodina=0;
31        do
32          {
33            balans *= intRata;
34            ++brojGodina;
35          }
36        while (balans < ciljniBalans);
37        Console.WriteLine("Za {0} godin{1} imate balans od {2}.",
38            brojGodina, brojGodina==1 ? "u": "a", balans);
39      }
40    }
```

Ovo će odbaciti vrednosti koje nemaju smisla, tako da ćemo dobiti sledeće:



Ispravnost korisnikovog unosa je važna tema kada dođe do dizajna aplikacije.

Petlje for

Poslednja vrsta petlji koju ćemo videti u ovom predavanju jeste petlja for, koja se izvršava unapred određeni broj puta i održava svoj sopstveni brojač. Da bi se definisala for petlja, trebaju nam sledeće informacije:

- Početna vrednost za inicijalizaciju brojačke promenljive;
- Uslov za nastavak petlje, koji uključuje brojačku promenljivu;
- Operacija koja se izvršava nad brojačkom promenljivom na kraju svakog ciklusa petlje.

Na primer, ako želimo petlju sa brojačem koji se povećava od 1 do 10 u koracima po jedan, početna vrednost je 1 - uslov je da brojač bude manji ili jednak 10, a operacija koja se izvodi na kraju svakog ciklusa je povećavanje brojača za jedan.
Ove informacije moraju biti smeštene u strukturu for petlje na sledeći način:

```
for (<inicijalizacija>; <uslov>, <operacija>)
{
    <kod unutar petlje>
}
```

Ovo radi na isti način kao sledeća petlja while:

```
(<inicijalizacija>
while (<uslov>)
{
    <kod unutar petlje>
    <operacija>
}
```

Međutim, format petlje for čini k6d čitljivijim, zato što sintaksa uključuje specifikaciju kompletne petlje na jednom mestu, umesto da je podeljena na vise različitih iskaza u različitim delovima koda.
Ranije smo koristili petlje while i do da bismo ispisivali brojeve od 1 do 10. Pogledajmo kakav kod je potreban da bi se to izvelo sa petljom for:

```
int i;
for (i=1; i<=10; ++ i)
{
    Console.Writeline(“{0}“, i);
}
```

Brojačka promenljiva je celobrojna vrednost sa imenom i, početne vrednosti 1, i povećava se za 1 na kraju svakog ciklusa. Tokom svakog ciklusa vrednost i se ispisuje na konzoli.
Primetićete da se kod zaustavlja kada i dostigne vrednost 11. To je zato što se na kraju ciklusa u kome je i jednako 10 , i poveća na 11. To se događa pre obrade uslova i<=10 kada se petlja i završava.
Kao i kod petlji **while**, i petlje **for** se izvršavaju samo kada je vrednost uslova **true** pre prvog ciklusa, tako da kod u petlji ne mora ni da se izvrši.
Konačno, možemo deklarisati brojačku promenljivu kao deo for iskaza, tako što ćemo prepravi" kod od malopre:

```
for (int i=1; i<=10; ++ i)
{
    Console.Writeline(“{0}“, i);
}
```

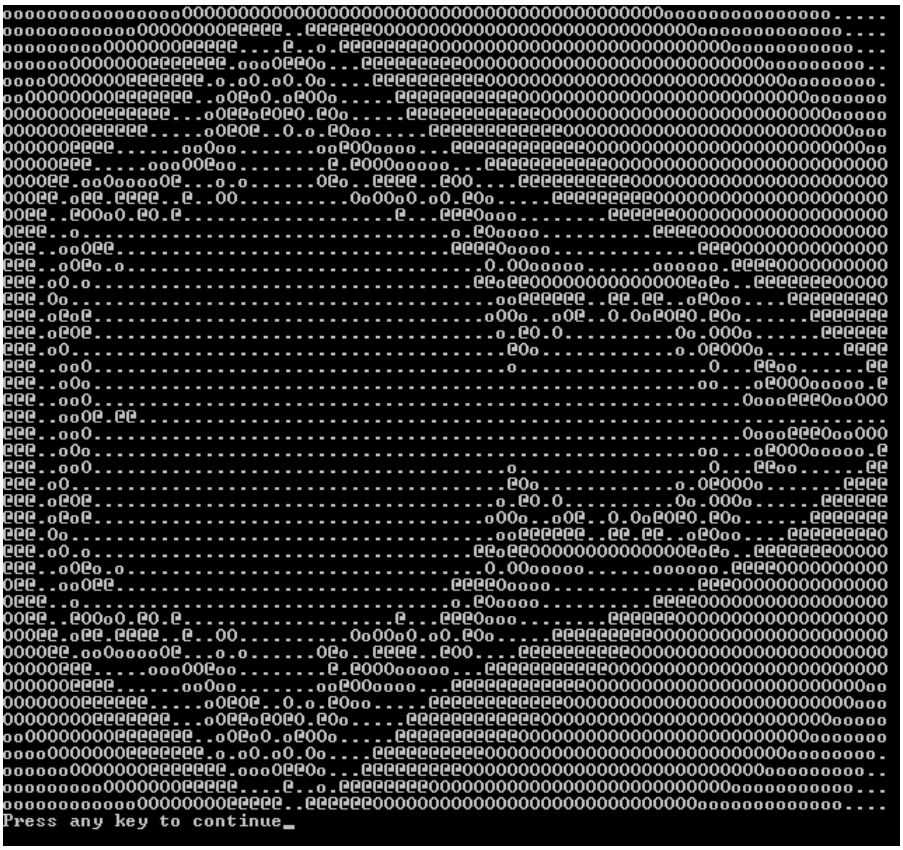
Mada, ako ovo uradimo, promenljiva i neće biti dostupna izvan ove petlje. Pogledajmo primer korišćenja petlji for. Pošto smo do sada prilično koristili petlje, učiniću ovaj primer malo interesantnijim: prikazaće Mandelbrotov skup (koristeći obične tekstualne znakove, tako da neće izgledati spektakularno!).

Vežba br. 8.

Napravite novi projekat konzolne aplikacije birajući File | New | Project... iz menija:
Izaberite Visual C# Projects direktorijum unutar prozora Project Types: i tip projekta Console Application u okviru prozora Templates: (za ovo ćete morati malo da pomerite prozor na dole).U okviru za tekst Location,promenite putanju u **C:\Temp\SoftIng\LecturesCode\Vezba8** (ovaj će direktorijum biti automatski napravljen ukoliko već ne postoji), i ostavite podrazumevani tekst u okviru za tekst Name: **IskazFor.**
Dodajte sledeći kod u Class1.cs:

```
14      static void Main(string[] args)
15      {
16        double realKoord, imagKoord;
17        double realPom, imagPom, realPom2, arg;
18        int iteracija;
19        for (imagKoord = 1.2; imagKoord >= -1.2; imagKoord -= 0.05)
20        {
21          for (realKoord = -0.6; realKoord <= 1.77; realKoord += 0.03)
22          {
23            iteracija=0;
24            realPom = realKoord;
25            imagPom = imagKoord;
26            arg = (realKoord*realKoord)+(imagKoord*imagKoord);
27
28            while ((arg < 4) && (iteracija < 40))
29            {
30              realPom2 = (realPom*realPom)-(imagPom*imagPom)-realKoord;
31              imagPom = (2*realPom*imagPom)-imagKoord;
32              realPom=realPom2;
33              arg = (realPom*realPom)+(imagPom*imagPom);
34              iteracija+=1;
35            }
39            switch (iteracija % 4)
40            {
41              case 0:
42                Console.Write(".");
43                break;
44              case 1:
45                Console.Write("o");
46                break;
47              case 2:
48                Console.Write("O");
49                break;
50              case 3:
51                Console.Write("@");
52                break;
53            }
54          }
55          Console.Write("\n");
56        }
57      }
58      }
59  }
```

Pokrenite kod



Kako to radi

Nećemo ulaziti u detalje kako se radi izračunavanje Mandelbrotovog skupa, ali demo preci osnove, da bismo shvatili zašto nam trebaju petlje koje smo koristili. Slobodno preskočite sledeća dva paragrafa, ako vas ne interesuje matematika - bitno je shvatiti kako se ponaša kod.
Svaka pozicija u Mandelbrotovoj slici odgovara imaginarnom broju oblika N=x+y*i, kome je realni deo x, imaginarni deo y, a i je kvadratni koren iz -1. Koordinate x i y na slid odgovaraju x i y delu imaginarnog broja.

Za svaku poziciju na slici posmatramo argument N, što je kvadratni koren iz x*x + y*y. Ako je ova vrednost veća ili jednaka 2, onda za ovu poziciju kažemo da ima vrednost 0. Ako je N manje od 2 onda menjamo N u vrednost N*N-N (što nam daje N= (x*x-y*y* -x) + (2*x*y-y) *i). Opet proveravamo vrednost argumenta N, ako je ova vrednost veća ili jednaka 2 za nju kažemo da pozicija ovog broja odgovara vrednosti 1. Ovaj proces se nastavlja sve dok ne dodelimo vrednost poziciji na slici ili dok se ne izvrši određeni broj iteracija.

Na osnovu vrednosti dodeljene svakoj tački na slici, imali bismo, u grafičkom okruženju, mesto za tačku određene boje na ekranu. Međutim, pošto koristimo tekstualno okruženje , mi samo postavljamo znak na ekran.
Pogledajmo kod i koje petlje on sadrži. Počećemo od deklarisanja promenljivih koje će nam trebati za proračun:

```
double realKoord,  imagKoord;
double realPom,  imagPom,  realPom2,  arg;
int  iteracija;
```

Ovde su **realKoord** i **imagKoord** realni i imaginarni delovi N, dok će druge promenljive tipa double služiti za čuvanje privremenih informacija za vreme izračunavanja. Promenljiva **iteracija** pamti koliko nam iteracija treba da bi argument N(arg) bio 2 ili veći

Zatim pokrećemo dve for petlje, da bismo prošli kroz koordinate pokrivajući celu sliku (koristimo malo kompleksniju sintaksu za promenu brojača u odnosu na ++ ili —, uobičajenu i moćnu tehniku):

```
19        for (imagKoord = 1.2; imagKoord >= -1.2; imagKoord -= 0.05)
20        {
21          for (realKoord = -0.6; realKoord <= 1.77; realKoord += 0.03)
22          {
```

Izabrali smo odgovarajuća ograničenja da bismo prikazali glavni deo Mandelbrotovog skupa. Slobodno se igrajte sa ovim ako želite da isprobate zumiranje slike.
U okviru ove dve petlje imamo k6d koji se drži određene tačke u Mandelbrotovom skupu, dajući j nam vrednost za N da se pozabavimo. Ovde iterativno vršimo traženi proračun, koji nam daje j vrednost koju ispisujemo za tekuću tačku.
Prvo inicijalizujemo određene promenljive:

```
23            iteracija=0;
24            realPom = realKoord;
25            imagPom = imagKoord;
26            arg = (realKoord*realKoord)+(imagKoord*imagKoord);
27
```

Sledeća je petlja while koja izvodi našu iteraciju. Ovde koristimo petlju while umesto petlje do, za slučaj da je argument početne vrednosti N veći od 2, u kom slučaju je **iteracija**=0 , pa dalja izračunavanja nisu potrebna.
Upamtite da se ovde argument ne izračunava do kraja, već samo vrednost x*x +y*y koja se proverava da bi se videlo da li je vrednost manja od 4. Ovo pojednostavljuje računanje, pošto znamo da je 2 kvadratni koren iz 4 i nema potrebe da sami izračunavamo kvadratne korene:

```
28            while ((arg < 4) && (iteracija < 40))
29            {
30              realPom2 = (realPom*realPom)-(imagPom*imagPom)-realKoord;
31              imagPom = (2*realPom*imagPom)-imagKoord;
32              realPom=realPom2;
33              arg = (realPom*realPom)+(imagPom*imagPom);
34              iteracija+=1;
35            }
```

Maksimalni broj iteracija ove petlje, koji izračunava vrednosti kako je gore navedeno, je 40.
Kada dobijemo vrednosti za trenutnu tačku smeštenu u **iteracija**, koristimo iskaz switch da izaberemo znak za izlaz. Ovde koristimo samo 4 različita znaka, umesto 40 mogućih vrednosti, i koristimo operator modula (%) tako da vrednosti 0,4, 8 i tako dalje daju jedan znak, vrednosti 1, 5,9 i tako dalje daju drugi znak i slično:

```
39            switch (iteracija % 4)
40            {
41              case 0:
42                Console.Write(".");
43                break;
44              case 1:
45                Console.Write("o");
46                break;
47              case 2:
48                Console.Write("O");
49                break;
50              case 3:
51                Console.Write("@");
52                break;
53            }
```

Upamtite da koristimo metod Console.Write() umesto Console.WriteLine(), pošto ne želimo novi red svaki put kada se ispisuje znak.
Na kraju jedne od najviše uvučenih for petlji želimo da završimo red, tako da koristimo iskočnu sekvencu koju smo ranije upoznali kako bismo na izlaz poslali znak za novi red:

Console.Write("\n");

Rezultat ovoga je da se svaki red odvaja od sledećeg i poravnava u liniju na pravi način.
Konačni rezultat ove aplikacije, iako nije spektakularan, dovoljno je impresivan i prilično lepo prikazuje od kakve koristi nam mogu biti petlje i grananja.

Prekidanje petlji

S vremena na vreme zatreba nam finija kontrola nad kodom koji se izvršava u petlji. Jezik C# obezbeđuje četiri komande koje nam pomažu u tome. Tri od njih smo već videli u dragim situacijama:

- **break**-trenutno završava petlju
- **continue** - trenutno zaustavlja ciklus petlje (izvršavanje se nastavlja od sledećeg ciklusa)
- **goto** - dozvoljava iskakanje iz petlje na obeleženo mesto u kodu (nije preporučljivo ako želite da vam kod bude čitljiv)
- **return** - iskače iz petlje i iz funkcije koja je sadrži.

Komanda **break** jednostavno izlazi iz petlje i nastavlja izvršenje prvog reda koda posle petlje, na primer:

```
int  i  =  1;
while  (i  <=10);
{
   if  (i  ==6)
   break;
   Console.WriteLine(“{0}“, i++);
}
```

Kod će ispisati brojeve od 1 do 5, pošto će komanda break prouzrokovati izlaz kada i bude jednako 5.

continue zaustavlja samo trenutni ciklus, a ne celu petlju, na primer:

```
int i = 1;
for (i=l; i<=10; i++) ;
{
    if ( (i. % 2)==0)
        continue;
    Console.WriteLine(i) ;
}
```

Treći metod je korišćenje iskaza goto kao što smo videli ranije, na primer:

```
int i = 1;
while (i <=10) ;
{
    if (i == 6)
    goto TackaIzlaza;
    Console.WriteLine (" (0) ", i++) ;
}
Console.WriteLine ("Ovaj kod se nikad neće izvršiti . ");
TackaIzlaza:
Console.WriteLine ("Ovaj kod se izvršava kad se iz petlje izađe uz pomoć goto. ");
```

Upamtite da je izlazak iz petlje sa goto dozvoljen (iako je to neuredno), ali nije dozvoljeno uskakanje u petlju spolja.

Beskonačne petlje

Moguće je (greškom ili namerno), definisati petlje koje se nikada ne završavaju, takozvane beskonačne petlje. Kao vrlo jednostavan primer pogledajte sledeće:

```
while (true)
{
    // kod u petlji
}
```

Ova situacija može biti korisna ponekad, a iz takvih petlji uvek možemo izaći pomoću iskaza break.
Međutim, kada se ovo dogodi slučajno, može biti vrlo neprijatno. Pogledajte sledeću petlju koja je slična for petlji u zadnjem delu:

```
int i = 1;
while (i <=10);
{
    if ((i % 2)==0)
        continue;
    Console.WriteLine{"{0}", i++);
}
```

Ovde se i ne povećava sve dok ne dođe do zadnjeg reda u petlji, koja se nalazi posle continue iskaza. Ako se dođe do continue iskaza (što se događa ako je i jednako 2) sledeći ciklus će koristiti istu vrednost za i, nastavljajući petlju, ispitujući istu vrednost i, i tako dalje. Ovo će prouzrokovati zamrzavanje aplikacije. Upamtite da je i iz takvih aplikacija moguće izaći na normalan način, tako da nema potrebe da resetujete računar ako se to dogodi.

Sažetak

U ovom predavanju, proširili smo naše programersko umeće, koristeći različite strukture koje možemo primeniti u našem kodu. Pravilna upotreba ovih struktura je od velikog značaja kada počnemo da pravimo kompleksnije aplikacije, a vremenom ćemo ih videti u ovoj knjizi.

Prvo smo se upoznali sa Bulovom logikom, ako se osvrnemo unazad, nakon što smo prošli kroz celo predavanje postaje jasno da su naše početne pretpostavke bile tačne, tj. da je ova tema veoma bitna kada se radi o implementaciji grananja i petlji. Vrlo je važno dobro se upoznati sa operatorima i tehnikama koje su ovde opisane.

Grananje nam omogućava da uslovno izvršavamo kod, što kad se kombinuje sa petljama, stvara kombinovane strukture u našem C# kodu. Kada imate petlje unutar petlji, unutar iskaza if unutar petlji, počinjete da shvatate zašto je uvlačenje koda bitno. Ako sav kod pomerimo uz levu ivicu ekrana, odmah postaje težak za čitanje i još teži za otklanjanje grešaka. Ako ste shvatili I uvlačenja redova u kodu - cenićete to znanje kasnije. VS će uraditi dosta toga za nas, ali nije lose da se tokom pisanja još malo uvlače redovi.

Vežbe

1. Ako imamo dve celobrojne vrednosti smeštene u promenljive var 1 i var2, koji logički test možemo izvesti da bismo videli da li je jedna ili druga (ali ne obe) veća od 10?

2. Napišite aplikaciju koja uključuje logiku iz vežbe 1, i koja dobija dva broja od korisnika i prikazuje ih, ali odbacuje bilo koji unos koji je veći od 10 i tad traži dva nova broja.

3. Sta nije u redu sa sledećim kodom?

```
int  i;
for   (i=1;i<=10;i++)
{
    if    ((i % 2) = 0)
        continue;
    Console.WriteLine (i) ;
}
```

4. Izmenite aplikaciju Mandelbrotovog skupa tako da zatraži granice slike od korisnika i prikaže odabrani deo slike. Trenutni izlaz koda prikazuje onoliko znakova koliko može stati u jedan red konzolne aplikacije, pa vodite računa o tome da svaka slika može stati u isti prostor, da bi se povećao vidljivi prostor.

Funkcije

Kod koji smo do sada videli uglavnom je bio smešten unutar jednog bloka, eventualno sa nešto petlji da bi se deo koda ponovio, ili grananja radi uslovnog izvršavanja iskaza. Izvođenje operacije nad podacima podrazumevalo je postavljanje koda na tačno određeno mesto gde on treba da funkcioniše.

Takva programska struktura je ograničena. Često ćemo nailaziti na zadatke (kao što je pronalaženje najveće vrednosti unutar niza) koji se moraju izvršiti nekoliko puta tokom izvršenja programa. Možemo postaviti identične (ili približno identične) delove koda u aplikaciju tamo gde je to potrebno, ali mogu se pojaviti problemi. Ako napravimo najmanju izmenu u takvom delu koda koji se tiče nekog standardnog postupka (npr. ispravljanje greške u kodu), taj deo koda moramo izmeniti na svim ostalim mestima unutar aplikacije. Ukoliko samo jedno od njih propustimo da ispravimo, posledice mogu biti dramatične i dovesti do toga da kompletna aplikacija ne radi. Pored toga, sama aplikacija bi bila vrlo velika.
Ovaj problem rešavamo uz pomoć funkcija. Funkcije u jeziku C# predstavljaju sredstva kojima se obezbeđuje blok koda koji može biti izvršen na bilo kom mestu u aplikaciji.

Funkcije određenog tipa koje se nalaze u ovom predavanju poznate su kao metode. Iako ovaj termin ima vrlo određeno značenje u .NET programiranju, za sada cento izbegavati njegovu upotrebu.

Na primer, funkciju koja pronalazi maksimalnu vrednost u nizu možemo koristiti na bilo kom mestu u programu i svaki put iskoristiti iste redove koda. Potrebno je da samo jednom napišemo kod, i svaka promena koju unesemo odraziće se na sva mesta na kojima se kod izvršava. To znači da funkcija sadrži kod koji se može **iznova upotrebljavati**.
Prednost funkcija je u tome što čine kod čitljivijim, a takođe ih možemo koristiti za grupisanje međusobno povezanog koda. Koristeći funkcije smanjujemo glavni deo koda u aplikaciji, zato što se sporedni delovi izvršavaju izvan glavnog. To podseća na sabijanje velikih delova koda u VS-u unutar prozora outline view, a daje i logičniju struktura aplikaciji.

Funkcije se takođe mogu koristiti pri pravljenju **višenamenskog** koda, jer one mogu izvršiti iste operacije nad različitim podacima. Funkciju možemo snabdeti potrebnim informacijama u obliku
parametara, a rezultate funkcija možemo dobiti u obliku povratnih vrednosti. Za gornji primer, parametar bi bio niz koji se pretražuje, dok bi povratna vrednost bila maksimalna vrednost unutar niza. To znači da istu funkciju možemo koristiti za rad sa različitim nizovima. Parametri i povratne vrednosti funkcije zajedno definišu **potpis** funkcije.

U ovom predavanju ćemo videti na koji način se definišu proste funkcije koje niti primaju niti vraćaju podatke. Zatim ćemo videti kako se podaci prenose u funkciju i iz nje.
Nakon toga, obradićemo temu **opsega važenja promenljive**. Ova tema se bavi načinom na koji se podaci u C#-u lokalizuju u posebne delove koda - što dobija na značaju kada delimo kod na vise različitih funkcija.
Upoznaćemo se sa vrlo važnom funkcijom u C# aplikacijama - **Main().** Videćemo kako da iskoristimo već ugrađene modele ove funkcije, da bismo iskoristili **argumente iz komandne linije**, koji nam dozvoljavaju transfer informacija u aplikacije kada se one pokrenu.
Pogledaćemo dodatne mogućnosti struktura pomoću kojih funkcije predstavljamo kao članove struktura. Na kraju ćemo upoznati dve naprednije teme: preopterećivanje funkcije i delegate. Preopterećivanje funkcije je tehnika koja se koristi za stvaranje različitih funkcija sa istim imenom ali različitim potpisom. Delegat je tip promenljive koji nam dozvoljava indirektno korišćenje funkcije. Jedan delegat se može koristiti za pozivanje bilo koje funkcije koja ima određeni potpis, što nam omogućava da biramo između različitih funkcija u toku izvršenja programa.

Definisanje i korišćenje funkcija

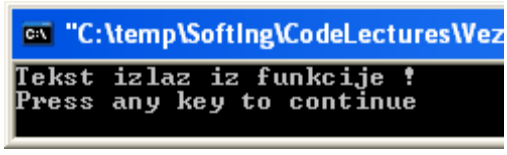
U ovom delu ćemo videti kako da aplikaciji dodamo funkciju i zatim je koristimo (pozivamo) iz koda. Počećemo od osnova, sa prostom funkcijom koja ne menja podatke sa kodom koji je poziva; zatim ćemo videti napredniju upotrebu funkcija.

Vežba br. 9.

Napravite novi projekat konzolne aplikacije birajući File | New | Project... iz menija:
Izaberite Visual C# Projects direktorijum unutar prozora Project Types: i tip projekta Console Application u okviru prozora Templates: (za ovo ćete morati malo da pomerite prozor na dole). U okviru za tekst Location;promenite putanju u **C:\Temp\SoftIng\LecturesCode\Vezba9** (ovaj će direktorijum biti automatski napravljen ukoliko već ne postoji), i ostavite podrazumevani tekst u okviru za tekst Name: **PrimerFunkcije1**
Dodajte sledeći kod u Class1.cs:

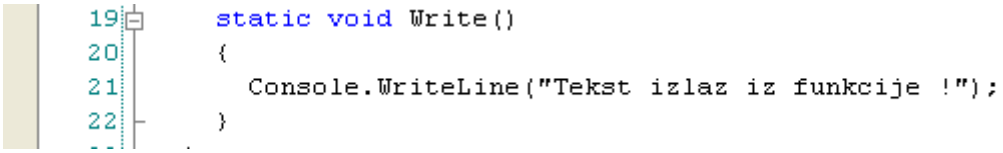
```
8   class Class1
9   {
10    /// <summary>
11    /// The main entry point for the application.
12    /// </summary>
13    [STAThread]
14    static void Main(string[] args)
15    {
16      Write();
17    }
18
19    static void Write()
20    {
21      Console.WriteLine("Tekst izlaz iz funkcije !");
22    }
23  }
24 }
25
```

Pokrenite program birajući **Debug | Start Without Debugging**



Kako to radi

Sledeća četiri reda vašeg koda definišu funkciju pod imenom **Write()** :



Ovaj kod jednostavno ispisuje tekst u konzolni prozor. Od toga su trenutno bitniji mehanizmi koji stoje iza definicije, kao i upotrebe funkcije.Definicija funkcije se sastoji iz sledećeg:

- dve ključne reči - **static** i **void**,
- imena funkcija koje prate zagrade **Write()**,
- bloka koda koji se izvršava uokviren zagradama.

Kod koji definiše funkciju Write() vrlo je sličan dragim redovima koda u aplikaciji:

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

Sav kod koji smo do sada napisali (izuzev definicija tipova) bio je deo jedne funkcije. Funkcija Main() je ulazna tačka za konzolnu aplikaciju. Kada se C# aplikacija izvršava, poziva se funkcija koja sadrži ulaznu tačku. Kada se ta funkcija izvrši, aplikacija se završava. Sav izvršni kod u jeziku C# mora imati ulaznu tačku.

Jedina razlika između funkcije Main() i funkcije Write() (izuzev redova koda koji se nalaze u njima) jeste u tome što unutar zagrada posle imena funkcije Main postoji neki kod. Na taj način se navode parametri o kojima ćemo uskoro detaljnije govoriti.

Kako je gore pomenuto, Main () i Write () su definisane uz pomoć ključnih reči **static** i **void**. Ključna reč **static** odnosi se na objektno orijentisane koncepte. Za sada jedino treba zapamtiti da sve funkcije koje ćemo ovde koristiti u aplikacijama mogu koristiti ovu ključnu reč.
Sa druge strane, ključnu reč **void** je mnogo jednostavnije objasniti. Ona ukazuje na to da funkcija ne vraća nikakvu vrednost. Kasnije ćemo videti šta treba navesti ukoliko funkcija vraća neku vrednost.

Kod koji poziva funkciju je sledeći:

Write() ;

Jednostavno unesemo ime funkcije sa praznim zagradama. Kada se u toku izvršenja programa naiđe na ovu tačku, izvršava se kod funkcije Write ().

Upamtite da su zagrade koje se koriste na oba mesta, prilikom definicije funkcije i dok je pozivamo, obavezne. Ukoliko ih uklonite, kod se neće kompajlirati.

Povratne vrednosti

Najjednostavniji način za razmenu podataka sa funkcijom jeste korišćenje povratne vrednosti. Funkcija koja ima povratnu vrednost izračunava je na isti način na koji se unutar izraza izračunava vrednost promenljive. Povratne vrednosti, kao i promenljive, imaju svoj tip.
Na primer, možemo imati funkciju sa imenom **getString()** čija je povratna vrednost tipa string. To možemo koristiti u kodu na sledeći način:

```
string mojString;
mojString = getString() ;
```

Možda ćemo imati funkciju pod imenom **getVal()** koja vraća vrednost tipa double, koju možemo koristiti u matematičkim izrazima:

```
double mojVal;
double multipler = 5.3;
mojVal = getVal()   * multipler;
```

Kada funkcija vraća vrednost moramo je izmeniti na dva načina:

- Moramo navesti tip povratne vrednosti u deklaraciji funkcije, umesto ključne reči **void**.
- Moramo koristiti ključnu reč **return** da bismo završili funkciju i preneli povratnu vrednost u pozivajući kod.

Prikazano preko koda za tip funkcije konzolne aplikacije koju smo već videli, to izgleda ovako:

```
Static  <returnType>  <functionName>()
{
   ..........
   ..........
   return <returnValue>;
}
```

Jedino ograničenje je u tome da <**returnValue**> mora biti vrednost koja je tipa <**returnType**>, ili može biti implicitno konvertovana u taj tip. Pa ipak, <**returnType**> može biti bilo koji tip, uključujući komplikovanije tipove koje smo već videli. To može biti jednostavno, kao na primer:

```
static double getVa()
{
    return 3.2;
}
```

Međutim, povratne vrednosti su obično rezultat proračuna izvedenog unutar funkcije, jer se to jednostavno može postići sa promenljivom tipa **const**.

Kada dođe do iskaza **return**, izvršenje programa se momentalno vraća na pozivajući kod. Redovi unutar funkcije koji se nalaze iza ovog iskaza neće biti izvršeni. To ipak ne znači da se **return** mora postaviti na kraj funkcije. Iskaz **return** možemo koristiti pre toga, možda nakon izvršenja nekog logičkog grananja.

Postavljanje iskaza return u for petlju, blok if, ili neku sličnu struk-turu, dovodi do trenutnog završetka strukture i funkcije. Na primer:

```
static double getVal()
{
   double checkVal;
   // checkVal je dodeljena vrednost kroz neku logiku
   if (checkVal < 5)
        return 4.7;
   return 3.2;
}
```

Ovde može biti vraćena jedna od dve vrednosti, zavisno od vrednosti u **checkVal**.
Jedino ograničenje je da iskaz return mora biti izvršen pre nego što se dostigne krajnja oznaka funkcije **}**. Sledeće nije dozvoljeno:

```
static  double getVal()
{
   double  checkVal;
   //  checkVal  je dodeljena vrednost kroz  neku  logiku
   if   (checkVal < 5)
        return  4.7;
}
```

Ako je checkVal >= 5, onda se iskaz **return** neće izvršiti, što nije dozvoljeno. Sve putanje kroz funkciju se moraju završiti jednim iskazom return.
Iskaz return se može koristiti u funkcijama koje se deklarišu pomoću ključne reči void, koje nemaju povratnu vrednost. Ako to uradimo, funkcija će se zaustaviti. Kada koristimo iskaz return na ovaj način, pogrešno je navesti povratnu vrednost između ključne reči return i tačke-zareza iza nje.

Parametri

Kada funkcija treba da prihvati parametre, moramo navesti sledeće:

- listu parametara koje funkcija prihvata, zajedno sa njihovim tipovima unutar definicije funkcije,
- odgovarajuću listu parametara u svakom pozivu funkcije.

Podrazumeva se sledeći kod:

```
Static <returnType> <functionName>(<paramType> <paramName>,  ...)
{
    ........
    ........
    return <returnValue>;
}
```

Ovde možemo imati proizvoljan broj parametara, svaki sa tipom i imenom. Parametri su odvojeni zarezima. Svakom od parametara se može pristupiti unutar funkcije kao promenljivoj.

Na primer, prosta funkcija može imati dva parametra tipa double i vratiti njihov proizvod:

```
static double product   (double param1, double param2)
{
    return param1 * param2;
}
```

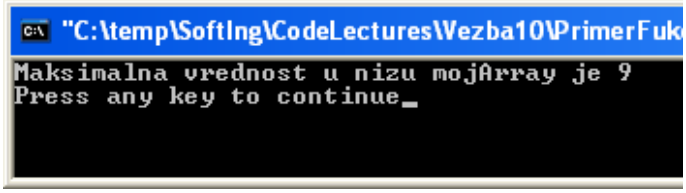
Pogledajmo malo složeniji primer.

Vežba br. 10.

Napravite novi projekat konzolne aplikacije birajući File | New | Project... iz menija:
Izaberite Visual C# Projects direktorijum unutar prozora Project Types: i tip projekta Console Application u okviru prozora Templates: (za ovo ćete morati malo da pomerite prozor na dole). U okviru za tekst Location; promenite putanju u **C:\Temp\SoftIng\LecturesCode\Vezba10** (ovaj će direktorijum biti automatski napravljen ukoliko već ne postoji), i ostavite podrazumevani tekst u okviru za tekst Name: **PrimerFunkcije2**
Dodajte sledeći kod u Class1.cs:

```
8   class Class1
9   {
10    /// <summary>
11    /// The main entry point for the application.
12    /// </summary>
13    [STAThread]
14    static void Main(string[] args)
15    {
16      int[] mojArray = {1,8,3,6,2,5,9,3,0,2};
17      int maxProm = MaxVrednost(mojArray);
18      Console.WriteLine("Maksimalna vrednost u nizu mojArray je {0}", maxProm);
19    }
20    static int MaxVrednost(int[] intArray)
21    {
22      int maxProm = intArray[0];
23      for (int i = 1; i < intArray.Length; i++)
24      {
25        if (intArray[i] > maxProm)
26          maxProm = intArray[i];
27      }
28      return maxProm;
29    }
30  }
31 }
```

Pokrenite program.



Kako to radi

Ovaj kod sadrži funkciju koja radi ono čemu smo se nadali u primeru funkcije u uvodu ovog predavanja: prihvata niz celih brojeva kao parametre i vraća najveću vrednost tog niza. Definicija funkcije je sledeća:

```
        static int MaxVrednost(int[] intArray)
        {
          int maxProm = intArray[0];
          for (int i = 1; i < intArray.Length; i++)
          {
            if (intArray[i] > maxProm)
              maxProm = intArray[i];
          }
          return maxProm;
        }
      }
    }
```

Funkcija MaxVrednost() ima samo jedan definisani parametar - niz tipa **int** pod imenom **intArray**. Takođe ima povratnu vrednost tipa **int**. Izračunavanje maksimalne vrednosti je jednostavno. Lokalnoj promenljivoj **maxProm** tipa **int** dodeljena je prva vrednost u nizu, a zatim se ova vrednost poredi sa svakim sledećim elementom u nizu. Ako element sadrži veću vrednost od **maxProm**, ova vrednost zamenjuje trenutnu vrednost u **maxProm**. Kada se petlja završi, **maxProm** sadrži najveću vrednost u nizu i vraća se pomoću iskaza return.

Kod u funkciji Main() deklariše i inicijalizuje prost celobrojni niz koji koristimo u funkciji **maxVrednost**():

int[] mojArray = {1, 8, 3, 6, 2, 5, 9, 3, 0, 2};

Poziv funkcije maxVrednost() se koristi da bi se dodelila vrednost promenljivoj maxProm tipa

int maxProm = maxVrednost (mojArray);

Sledeći korak je ispisivanje ove vrednosti na ekran korišćenjem komande Console.WriteLine():

Console.WriteLine("Maksimalna vrednost u nizu mojArray je {0}", maxProm);

Slaganje parametara

Kada pozivamo funkciju moramo složiti parametre tačno kao što je navedeno u definiciji funkcije, odnosno moramo složiti tipove parametara, njihov broj i redosled. To znači da se sledeća funkcija:

```
Static void myFunction(string myString,   double myDouble)
{
   .....
   .....
}
```

ne može se pozvati sa:

myFunction(2.6, "Hello");

Ovde pokušavamo da prosledimo vrednost tipa double kao prvi parametar, a vrednost tipa string kao drugi, što nije redosled po kome su parametri navedeni u definiciji funkcije.
Takođe ne možemo koristiti:

myFunction("Hello");

Ovde prosleđujemo samo jedan parametar tipa string, a zahtevaju se dva. Ako pokušamo da koristimo bilo koji od ova dva poziva, doći će do greške pri kompajliranju, jer nas kompajler primorava da složimo parametre u funkcijama koje koristimo.

Ako se vratimo našem primeru - to znači da funkciju **maxVrednost**() možemo koristiti samo za dobijanje maksimalne vrednosti u nizu celih brojeva. Ako zamenimo kod u funkciji Main() sledećim kodom:

```
static void Main(string[] args)
{
  double[] mojArray = {1,8,3,6,2,5,9,3,0,2};
  int maxProm = MaxVrednost(mojArray);
  Console.WriteLine("Maksimalna vrednost u nizu mojArray je {0}", maxProm);
}
```

kod se neće kompajlirati zato što je tip parametra pogrešan.

Kasnije u ovom predavanju, u delu koji se bavi preopterećenjem operatora, videćemo tehniku koja zaobilazi ovaj problem.

Nizovi parametara

C# nam dozvoljava da definišemo jedan (i samo jedan) poseban parametar za funkciju. Ovaj parametar, koji mora biti zadnji u listi parametara u definiciji funkcije, poznat je kao **niz parametara**. Nizovi parametara dozvoljavaju nam da pozivamo funkcije koristeći promenljivu količinu parametara, a definisani su ključnom reči **params**.

Nizovi parametara mogu biti korisni za pojednostavljivanje koda, jer ne moramo prosleđivati nizove iz našeg pozivajućeg koda. Umesto toga, prosleđujemo nekoliko parametara istog tipa, koji se nalaze u nizu koji možemo koristiti unutar funkcije.

Sledeći kod je potreban da bi se definisala funkcija koja koristi niz parametara:

```
static <returnType> <functionName>(<p1Type><p1Name>, . . . , params <type> [] <name>)
{
   .....
   .....
   return <returnValue>;
}
```

Ovu funkciju možemo pozvati sa:

```
<functionName>(<p1>,...,<val1><val2>,...)
```

Ovde su <val1>, <val2> i tako dalje, vrednosti tipa <type> koji je korišćen da bi se inicijalizovao niz <name>. Ne postoji ograničenje u broju parametara koje možemo navesti; jedino ograničenje je da su svi tipa <type>. Cak i ne moramo navoditi parametre.

Poslednja stavka čini nizove parametara posebno korisnim za navođenje dodatnih informacija koje će funkcije koristiti tokom izvršenja. Na primer, imamo funkciju pod imenom getWord (), koja uzima vrednost tipa string kao prvi parametar i vraća prvu reč u nizu znakova:

string firstWord = getWord("Ovo je rečenica.");

Ovde će promenljivoj firstWord biti dodeljena vrednost "Ovo" .
Možemo dodati parametar params u funkciju getWord(), što nam dozvoljava da vratimo alternativnu reč po njenom indeksu:

string firstWord = getWord("Ovo je rečenica.",2);

Ako pretpostavimo da smo počeli brojanje od 1 za prvu reč, rezultat će biti to da je promenljivoj **firstWord** dodeljena vrednost "**je**".

Takođe, možemo dodati mogućnost ograničenja broja znakova koji se vraćaju preko trećeg parametra, kojem se takođe može pristupiti preko parametra params:

string firstWord = getWord("Ovo je rečenica.",3,3);

Ovde će promenljivoj **firstWord** biti dodeljen string "**reč**".

Vežba br. 11.

Napravite novi projekat konzolne aplikacije birajući File | New | Project... iz menija:
Izaberite Visual C# Projects direktorijum unutar prozora Project Types: i tip projekta Console Application u okviru prozora Templates: (za ovo ćete morati malo da pomerite prozor na dole). U okviru za tekst Location; promenite putanju u **C:\Temp\SoftIng\LecturesCode\Vezba11** (ovaj će direktorijum biti automatski napravljen ukoliko već ne postoji), i ostavite podrazumevani tekst u okviru za tekst Name: **PrimerParametara.** Dodajte sledeći kod u Class1.cs:

```
15      static int sumProm (params int[] vals)
16      {
17        int sum = 0;
18        foreach (int prom in vals)
19        {
20          sum +=prom;
21        }
22        return sum;
23      }
24
25      static void Main(string[] args)
26      {
27        int sum =sumProm(1, 5, 2, 9, 8);
28        Console.WriteLine("Sumirana vrednost je = {0}", sum);
29      }
30    }
31  }
32
```

Pokrenite program

Kako to radi

U ovom primeru je funkcija **sumProm**() definisana pomoću ključne reči params, tako da prihvati proizvoljan broj parametara tipa **int** (i nijedne drage):

```
static int sumProm (params int[] vals)
{
   .......
   .......
}
```

Kod u ovoj funkciji iterativno prolazi kroz vrednosti u nizu vals, sabira vrednosti zajedno, vraćajući rezultat.
U funkciji Main() pozivamo funkciju sa pet celobrojnih parametara:

```
int sum = sumProm(1, 5, 2, 9, 8);
```

Međutim, ovu funkciju smo mogli pozvati bez i jednog, sa jednim, dva, ili stotinu celobrojnih parametara - ne postoji ograničenje u količini parametara koje možemo navesti.

Prenos parametara po referenci i vrednosti

Funkcije koje smo do sada naveli koristile su prenos parametara po vrednosti. Podrazumeva se da smo, koriste parametre, prosleđivali njihove vrednosti u promenljive koje koristi funkcija. Bilo koja promena nad promenljivom unutar funkcije nema nikakvog efekta na parametre navedene u pozivu funkcije. Na primer, razmotrite funkciju koja udvostručuje i prikazuje vrednost prosleđenog parametra:

```
static void DupliranaVred(int prom)
{
   prom *=2;
   Console.WriteLine("duplirana promenljiva = (0)", prom);
}
```

Ovde je parametar **prom** udvostručen unutar funkcije. Ako je pozovemo na sledeći način:

```
int mojBroj = 5;
Console.WriteLine("mojBroj = {0]", mojBroj);
DupliranaVred (mojBroj) ;
Console.WriteLine("mojBroj = (0) ", mojBroj);
```

- na konzoli će se pojaviti tekst:

```
mojBroj = 5
duplirana promenljiva = 10
mojBroj = 5
```

Pozivanje funkcije **DupliranaVred**() sa promenljivom **mojBroj** kao parametrom, ne utiče na vrednost **mojBroj** unutar bloka Main(), iako je parametar val koji joj je dodeljen udvostručen. Ako poželimo da se vrednost promenljive **mojBroj** promeni, eto problema. Možemo koristiti funkciju koja vraća novu vrednost za **mojBroj**, a pozivamo je sa:

```
int mojBroj =  5;
Console.WriteLine{"mojBroj =   {0)",   mojBroj);
mojBroj =  DupliranaVred(mojBroj) ;
Console.WriteLine("mojBroj =   (0)",   mojBroj);
```

Ovaj kod teško da je intuitivan i neće se izboriti sa promenom vrednosti za vise promenljivih koje se koriste kao parametri (jer funkcija ima samo jednu povratnu vrednost).
Umesto toga, želimo da prosledimo parametar po **referenci**. To znači da će funkcija raditi sa potpuno istom promenljivom kao u pozivu funkcije, a ne samo sa promenljivom koja ima istu vrednost. Sve promene koje se dešavaju toj promenljivoj odraziće se na promenljivu koja se koristi kao parametar. Da bi se ovo uradilo, treba jednostavno upotrebiti ključnu reč **ref** za navođenje parametra:

```
static void DupliranaVred (ref  int prom)
{
   prom  *=  2;
   Console.WriteLine("duplirana promenljiva = (0)", prom);
}
```

Ponovimo to u pozivu funkcije (to je obavezno, jer je parametar tipa **ref** deo potpisa same funkcije):

```
int mojBroj =  5;
Console.WriteLine("mojBroj =   (0)",   mojBroj);
DupliranaVred (ref mojBroj);
Console.WriteLine("mojBroj =   (0}",   mojBroj);
```

Tada bi izlazni tekst na konzoli bio sledeći:

```
mojBroj = 5
duplirana promenljiva = 10
mojBroj = 10
```

Ovoga puta je promenljiva **mojBroj** izmenjena od strane funkcije **DupliranaVred**(). Postoje dva ograničenja u vezi sa promenljivom koja se koristi kao **ref** parametar. Prvo - funkcija može prouzrokovati promenu vrednosti u referentnom parametru, tako da moramo koristiti promenljivu koja nije konstanta u okviru poziva funkcije. Zbog toga nije dozvoljeno sledeće:

```
const int mojBroj = 5;
Console.WriteLine("mojBroj = (0)", mojBroj);
DupliranaVred (ref mojBroj) ;
Console.WriteLine("mojBroj = (0)", mojBroj);
```

Drago - promenljiva koju koristimo mora"biti inicijalizovana. Jezik C# nam ne dozvoljava pretpostavku da će parametar **ref** biti inicijalizovati u funkciji koja ga upotrebljava. Sledeći kod nije dozvoljen:

```
int mojBroj
DupliranaVred (ref mojBroj);
Console.WriteLine("mojBroj = {0}", mojBroj);
```

Izlazni parametri

Možemo definisati da je dati parametar izlazni parametar, preko ključne reči **out** koja se koristi na isti način kao i ključna reč **ref** (kao modifikator parametra u definiciji funkcije, i pri njenom pozivanju). Rezultat toga je isto ponašanje kao i kod referentnog parametra, u smislu da se vrednost parametra na kraju funkcije vraća promenljivoj koja je upotrebljena u njenom pozivu. Međutim, postoje i važne razlike. Dok za parametre tipa **ref** nije dozvoljeno koristiti promenljive kojima nije dodeljena vrednost, kod parametara **out** to možemo raditi. Pored toga, **out** parametar se mora tretirati kao da mu nije dodeljena vrednost od strane funkcije koja ga koristi. To znači da, iako je dozvoljeno koristiti promenljivu kojoj je dodeljena vrednost kao izlazni parametar, vrednost koju ona sadrži u pozivu funkcije biće izgubljena nakon izvršenja te funkcije.

Na primer, razmotrite proširenje funkcije MaxVrednost() koju smo ranije videli, a koja vraća najveću vrednost u nizu. Izmenićemo funkciju tako da vrati indeks elementa sa najvećom vrednosti unutar niza. Da bi bilo jednostavnije, vratićemo samo indeks prve pojave najvećeg elementa, ukoliko ih ima vise. Da bismo ovo uradili dodajemo izlazni parametar menjajući funkciju na sledeći način:

```
static int MaxVrednost(int[] intArray, out in maxIndex)
{
    int maxProm = intArray [0];
    maxIndex =0;
    for (int i = 1; i < intArray.Length; i++)
    {
        if (intArray[i] > maxProm)
        {
            maxProm = intArray [i] ;
            maxIndex = i;
        }
    return maxProm;
}
```

Možemo koristiti funkciju na sledeći način:

```
int []   myArray =   (1, 8, 3, 6, 2, 5, 9, 3, 0, 2);
int maxIndex
Console.Write("Maksimalna vrednost u nizu myArray je {0)", MaxVrednost(myArray,
out maxIndex);
Console.WriteLine{"Maksimalna vrednost niza je (0) element u nizu",  maxindex+1);
```

Rezultat ovoga je:

```
Maksimalna vrednost u nizu myArray je 9
Maksimalna vrednost niza je 7 element u nizu
```

Ovde je važno naglasiti da se ključna reč **out** mora upotrebiti u pozivu funkcije, baš kao i ključna reč **ref**.

Upamtite daje vrednosti maxIndex dodato jedan, i vraća se ovde dok je ispisujemo na ekran. To je zato da bismo indeks preveli u razumljiviji oblik, tako da se prvi element u nizu ne označava sa 0 već sa 1.

Opseg važenja promenljive

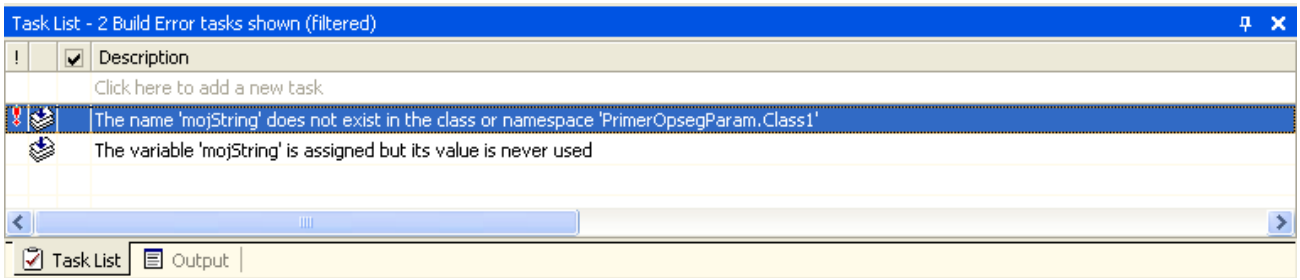
Možda ste se upitali zašto je uopšte potrebna razmena podataka sa funkcijama. Razlog je taj što se promenljivama u C#-u može pristupiti samo iz lokalnih regiona koda. Za datu promenljivu kaže se da ima opseg važenja unutar koga joj se može pristupiti.
Opseg važenja promenljive je važna tema i najbolje ćemo je predstaviti preko primera.
Definisanje i korišćenje osnovnih funkcija.

Vežba br. 12.

Napravite novi projekat konzolne aplikacije birajući File | New | Project... iz menija:
Izaberite Visual C# Projects direktorijum unutar prozora Project Types: i tip projekta Console Application u okviru prozora Templates: (za ovo ćete morati malo da pomerite prozor na dole). U okviru za tekst Location; promenite putanju u **C:\Temp\SoftIng\LecturesCode\Vezba12** (ovaj će direktorijum biti automatski napravljen ukoliko već ne postoji), i ostavite podrazumevani tekst u okviru za tekst Name: **PrimerOpsegParam.** Dodajte sledeći kod u Class1.cs:

```
14      static void Write()
15      {
16        Console.WriteLine("mojString = {0}", mojString);
17      }
18      static void Main(string[] args)
19      {
20        string mojString = "String koji je definisan u f-ji Main()";
21        Write();
22      }
23    }
24  }
25
```

Kompajlirajte kod i obratite pažnju na grešku i upozorenje koji se pojavljuju u listi zadataka:



Kako to radi

Sta je krenulo loše? Promenljivoj mojString, definisanoj u glavnom delu programa (u funkciji Main ()), nije moguće pristupiti iz funkcije Write ().
Razlog za ovu nepristupačnost jeste to što promenljiva ima svoje područje važenja unutar koga je validna. To područje podrazumeva blok koda u kom je deformisana i bilo koje direktno ugneždene blokove koda. Blokovi koda u funkcijama odvojeni su od blokova koda iz kojih se pozivaju.

Unutar funkcije Write() ime mojString je nedefinisano, pa je i promenljiva mojString deformisana u funkciji Main() **van opsega važenja** - ona se može koristiti samo u funkciji Main().

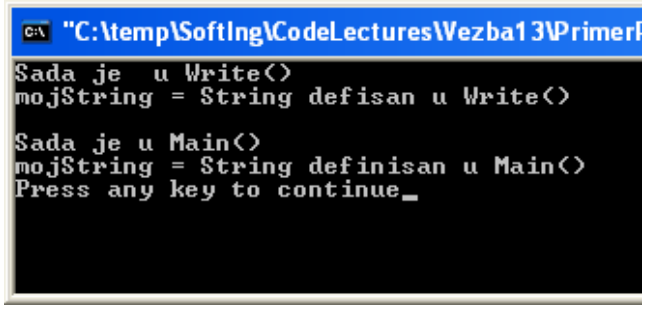
Možemo imati potpuno nezavisnu promenljivu u funkciji Write(), nazvanu mojString. Pokušajte da izmenite kod na sledeći način:

Vežba br. 13.

Napravite novi projekat konzolne aplikacije birajući File | New | Project... iz menija:
Izaberite Visual C# Projects direktorijum unutar prozora Project Types: i tip projekta Console Application u okviru prozora Templates: (za ovo ćete morati malo da pomerite prozor na dole). U okviru za tekst Location; promenite putanju u **C:\Temp\SoftIng\LecturesCode\Vezba13** (ovaj će direktorijum biti automatski napravljen ukoliko već ne postoji), i ostavite podrazumevani tekst u okviru za tekst Name: **PrimerParam2.** Dodajte sledeći kod u Class1.cs:

```
14      static void Write()
15      {
16        string mojString = "String defisan u Write()";
17        Console.WriteLine("Sada je  u Write()");
18        Console.WriteLine("mojString = {0}", mojString);
19      }
20      static void Main(string[] args)
21      {
22        string mojString = "String definisan u Main()";
23        Write();
24        Console.WriteLine("\nSada je u Main()");
25        Console.WriteLine("mojString = {0}", mojString);
26      }
27    }
28  }
```

Ovaj kod će se kompajlirati i rezultat će biti:



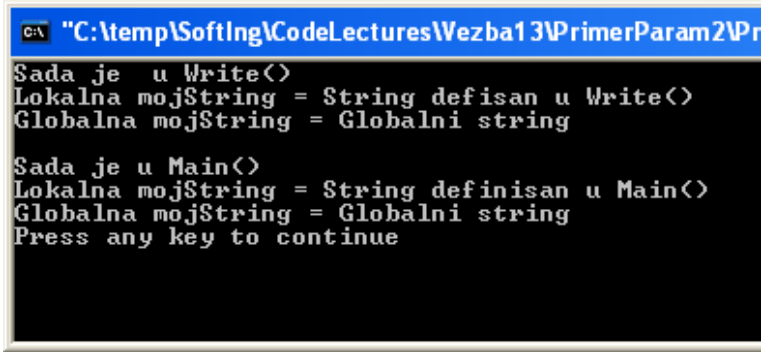
Operacije koje izvodi ovaj kod su sledeće:

- Main() definiše i inicijalizuje promenljivu tipa string sa imenom mojString.
- Main() prebacuje kontrolu na funkciju Write().
- Write() definiše i inicijalizuje promenljivu tipa string sa imenom mojString, koja nije isto što i promenljiva mojString definisana u funkciji Main().
- Write() ispisuje niz znakova na konzolu koji sadrži vrednost mojString definisan unutar funkcije Write().
- Write() vraća kontrolu funkciji Main().
- Main() ispisuje na konzolu vrednost koju sadrži mojString definisan unutar funkcije Main().

Promenljive čiji opseg važenja na ovaj način pokriva samo jednu funkciju, poznate su kao **lokalne promenljive**. Takođe je moguće imati **globalne promenljive**, čiji opseg važenja pokriva više funkcija. Izmenite kod na sledeći način:

```
8   class Class1
9   {
10    /// <summary>
11    /// The main entry point for the application.
12    /// </summary>
13
14    static string mojString;
15    [STAThread]
16    static void Write()
17    {
18      string mojString = "String defisan u Write()";
19      Console.WriteLine("Sada je  u Write()");
20      Console.WriteLine("Lokalna mojString = {0}", mojString);
21      Console.WriteLine("Globalna mojString = {0}", Class1.mojString);
22    }
23    static void Main(string[] args)
24    {
25      string mojString = "String definisan u Main()";
26      Class1.mojString="Globalni string";
27      Write();
28      Console.WriteLine("\nSada je u Main()");
29      Console.WriteLine("Lokalna mojString = {0}", mojString);
30      Console.WriteLine("Globalna mojString = {0}", Class1.mojString);
31    }
32  }
33 }
34
```

Rezultat je sada:



Ovde smo dodali još jednu promenljivu pod imenom mojString, ovaj put na višem nivou hijerarhije imena. Ova promenljiva je definisana na sledeći način:

```
static string mojString;
```

Primetićete da opet koristimo ključnu reč **static**. Pomenimo da u konzolnim aplikacijama, pri navođenju globalnih promenljivih u ovom obliku, moramo koristiti ključne reči **static** ili **const**. Ako želimo da menjamo vrednost globalne promenljive, moramo navesti **static**, jer **const** zabranjuje menjanje vrednosti promenljive.

Da bismo napravili razliku između ove promenljive i lokalnih promenljivih istog imena u funkcijama **Main()** i **Write(),** moramo se obratiti promenljivoj koristeći puno kvalifikovano ime. Ovde se globalnoj promenljivoj obraćamo sa **Class1.mojString**. To je neophodno samo kada imamo globalne i lokalne promenljive sa istim imenom. Kada ne bi bilo lokalne promenljive mojString, globalnu promenljivu bismo mogli upotrebljavati jednostavno sa mojString umesto Class1.mojString. Kada imamo lokalnu promenljivu koja ima isto ime kao i globalna promenljiva, za globalnu se kaže da je skrivena.
Vrednost globalne promenljive postavljena je u funkciju Main() sa:

```
Class1.mojString =  "Globalni  string";
```

A unutar funkcije Write() pristupa joj se sa:

```
Console.WriteLine("Globalni mojString =  (0)",  Class1.mojString);
```

Sigurno se pitate zašto ne bismo ovu tehniku koristili za razmenu podataka sa funkcijama, umesto što dodajemo parametre kako smo ranije videli. Postoje situacije kada je ovakav način poželjniji za razmenu podataka, ali ih ima isto toliko (ako ne i vise) kod kojih to nije slučaj. Izbor - da li koristiti globalnu promenljivu, zasniva se na načinu na koji će se koristiti funkcija. Globalne promenljive uglavnom nisu pogodne za opšte namenske funkcije koje su sposobne da rade sa bilo kojim podacima koje obezbedimo, i nisu ograničene samo na podatke u specifičnim globalnim promenljivama. Kasnije ćemo detaljnije govoriti o tome.

Opseg važenja promenljive u drugim strukturama

Jedna od stvari koju smo objasnili u prošlom delu ima posledice na opseg važenja izvan funkcija. Opseg važenja promenljivih uključuje blok koda u kom su definisane, i bilo koje direktno ugneždene blokove koda. To se odnosi i na druge blokove koda, kao što su grananja i petlje. Razmotrite sledeći kod:

Vežba br. 14.

Napravite novi projekat konzolne aplikacije birajući File | New | Project... iz menija: Izaberite Visual C# Projects direktorijum unutar prozora Project Types: i tip projekta Console Application u okviru prozora Templates: (za ovo ćete morati malo da pomerite prozor na dole). U okviru za tekst Location; promenite putanju u **C:\Temp\SoftIng\LecturesCode\Vezba14** (ovaj će direktorijum biti automatski napravljen ukoliko već ne postoji), i ostavite podrazumevani tekst u okviru za tekst Name: **PrimerOpsegParam2.** Dodajte sledeći kod u Class1.cs:

```
8   class Class1
9   {
10    /// <summary>
11    /// The main entry point for the application.
12    /// </summary>
13    [STAThread]
14    static void Main(string[] args)
15    {
16      int i;
17      for (i = 0; i<10; i++)
18      {
19        string text = "Linija" + Convert.ToString(i);
20        Console.WriteLine("{0}", text);
21      }
22      Console.WriteLine("Poslednji text definisan u petlji: {0}", text);
23    }
24  }
25 }
26
```

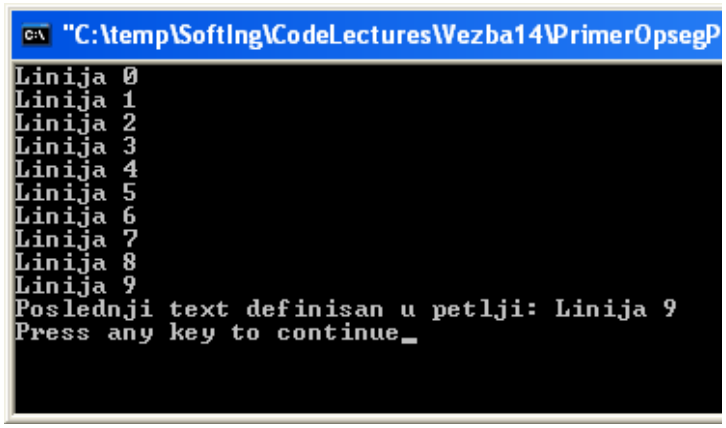
Ovde je promenljiva tipa string pod imenom **text** lokalna za for petlju. Taj kod se neće kompajlirati zato što poziv za **Console.WriteLine(),** koji se događa van petlje, pokušava da koristi promenljivu **text** koja je izvan opsega važenja kada je van petlje. Pokušajte da promenite kod na sledeći način:

```
8   class Class1
9   {
10    /// <summary>
11    /// The main entry point for the application.
12    /// </summary>
13    [STAThread]
14    static void Main(string[] args)
15    {
16      int i;
17      string text;
18      for (i = 0; i<10; i++)
19      {
20        string text = "Linija" + Convert.ToString(i);
21        Console.WriteLine("{0}", text);
22      }
23      Console.WriteLine("Poslednji text definisan u petlji: {0}", text);
24    }
25  }
26 }
```

Ovaj kod takođe neće raditi zbog toga što promenljiva mora biti deklarisana i inicijalizovana pre upotrebe, a text je inicijalizovana samo u for petlji. Vrednost dodeljena promenljivoj **text** gubi se kada se izađe iz petlje. Ipak, možemo napraviti sledeću promenu:

```
8   class Class1
9   {
10    /// <summary>
11    /// The main entry point for the application.
12    /// </summary>
13    [STAThread]
14    static void Main(string[] args)
15    {
16      int i;
17      string text = " ";
18      for (i = 0; i<10; i++)
19      {
20        text = "Linija " + Convert.ToString(i);
21        Console.WriteLine("{0}", text);
22      }
23      Console.WriteLine("Poslednji text definisan u petlji: {0}", text);
24    }
25  }
26 }
27
```

Promenljiva text će biti inicijalizovana van petlje pa možemo pristupiti njenoj vrednosti. Rezultat ovog jednostavnog koda je sledeći:



Ovde je poslednja vrednost, dodeljena promenljivoj text u petlji, dostupna i izvan petlje.
U svetlu prethodnih primera, nije odmah očigledno zašto promenljiva text posle petlje ne zadržava prazan niz znakova koji joj je dodeljen pre petlje.

Objašnjenje za to tiče se memorijske dodele za promenljivu text kao i bilo koje druge promenljive. Samim deklarisanjem osnovnog tipa promenljive ne postiže se mnogo. Tek kada se dodele vrednosti, one se raspoređuju u memoriji. Kada se ova raspodela dogodi unutar petlje, vrednost je deformisana kao lokalna i izlazi izvan opsega važenja kada je van petlje. Iako sama promenljiva nije lokalnog karaktera za petlju, vrednost koju sadrži jeste. Dodeljivanje vrednosti izvan petlje omogućava da ta vrednost postane lokalna za glavni deo koda, a da i dalje bude u opsegu unutar petlje. To znači da promenljiva ne izlazi iz svog opsega važenja, dokle god ne izađe iz bloka glavnog dela koda, tako da imamo pristup njenim vrednostima izvan petlje.

C# kompajler će otkriti probleme vezane za opseg važenja promenljivih i reagovaće poruka o grešci, koja nam pomaže da shvatimo problem opsega važenja promenljivih.
Uopšteno, vredi deklarisati i inicijalizovati sve promenljive pre nego što ih iskoristimo u bilo kom bloku koda. Izuzetak je deklarisanje promenljivih za petlju kao deo bloka unutar petlje, na primer:

```
for (int i+0; i<10; i++)
{
   .....
   .....
{
```

Ovde je promenljiva i lokalizovana na blok koda u petlji, ali to prihvatljivo, jer će nam ovaj brojač retko zatrebati u kodu izvan petlje.

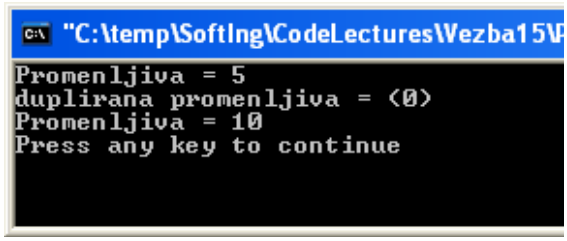
Parametri i povratne vrednosti nasuprot globalnim podacima

U ovom delu ćemo bliže pogledati razmenu podataka sa funkcijama preko globalnih podataka, i preko parametara i povratnih vrednosti. Da se podsetimo - govorimo o razlici izmedu koda kao što je:

Vežba br. 15.

Napravite novi projekat konzolne aplikacije birajući File | New | Project... iz menija:
Izaberite Visual C# Projects direktorijum unutar prozora Project Types: i tip projekta Console Application u okviru prozora Templates: (za ovo ćete morati malo da pomerite prozor na dole). U okviru za tekst Location; promenite putanju u **C:\Temp\SoftIng\LecturesCode\Vezba15** (ovaj će direktorijum biti automatski napravljen ukoliko već ne postoji), i ostavite podrazumevani tekst u okviru za tekst Name: **PrimerPovrVred.** Dodajte sledeći kod u Class1.cs:

```
8   class Class1
9   {
10    /// <summary>
11    /// The main entry point for the application.
12    /// </summary>
13    [STAThread]
14    static void DupliranaVred (ref int prom)
15    {
16      prom  *= 2;
17      Console.WriteLine("duplirana promenljiva = {0}", prom);
18    }
19    static void Main(string[] args)
20    {
21      int prom =  5;
22      Console.WriteLine("Promenljiva = {0}", prom);
23      DupliranaVred (ref prom);
24      Console.WriteLine("Promenljiva = {0}", prom);
25    }
26  }
27 }
28
```



Ovaj kod se malo razlikuje od koda koji smo videli ranije, kada smo koristili promenljivu sa imenom mojBroj u funkciji Main(). To je primer da lokalne promenljive mogu imati ista imena i da pritom ne smetaju jedna drugoj. To takođe znači da su ova dva primera koda slična, što nam dopusta da se fokusiramo na specifikacije razlike, ne vodeći brigu o imenima promenljivih.

```
 8   class Class1
 9   {
10     static int prom;
11     /// <summary>
12     /// The main entry point for the application.
13     /// </summary>
14     [STAThread]
15     static void DupliranaVred ()
16     {
17       prom  *= 2;
18       Console.WriteLine("duplirana promenljiva = {0}", prom);
19     }
20     static void Main(string[] args)
21     {
22       prom =  5;
23       Console.WriteLine("Promenljiva = {0}", prom);
24       DupliranaVred ();
25       Console.WriteLine("Promenljiva = {0}", prom);
26     }
27   }
28 }
29
```

Rezultat ovih funkcije je identičan.

Ne postoji čvrsto pravilo koje će odrediti koju od ove dve metode da koristimo, jer su obe tehnike savršeno ispravne. Pa ipak, ima nekoliko saveta koje treba razmotriti.
Za početak - showDouble() verzija koja koristi globalnu vrednost uvek će koristiti globalnu promenljivu val. Da bismo koristili ovu verziju, moramo koristiti globalnu promenljivu. To ograničava raznolikost funkcija i znači da stalno moramo kopirati vrednost globalne promenljive u druge promenljive ako hoćemo da smestimo rezultate. Pored toga, globalni podaci mogu biti modifikovani iz nekog drugog koda u aplikaciji, što može prouzrokovati nepredvidive rezultate (vrednosti bi se mogle promeniti, a mi bismo kasno za to saznali).
Ovaj gubitak raznolikosti često može biti dobitak. Biće trenutaka kada ćemo poželeti da koristimo funkciju samo za jednu namenu, a korišćenje globalnih podataka smanjuje mogućnost greške u pozivu funkcije (možda dodavanjem pogrešne promenljive).
Naravno, može se govoriti o tome da ova jednostavnost čini naš kod manje razumljivim. Eksplicitno navođenje parametara dozvoljava nam da na prvi pogled vidimo šta se menja. Ako vidimo poziv koji je, recimo, **myFunction** (var1, out var2), odmah znamo da su var1 i var2

važne promenljive koje ćemo uzeti u obzir, kao i da će promenljivoj var2 biti dodeljena nova vrednost kada se funkcija izvrši. Suprotno tome, ako funkcija ne prima nikakve parametre, ne možemo pretpostaviti kojim se podacima manipuliše unutar nje.

Treba zapamtiti da nije uvek moguće koristiti globalne podatke. Kasnije u knjizi ćemo video kod koji je napisan u različitim datotekama i/ili pripada različitim imenovanim prostorima koji komuniciraju između sebe preko funkcija. U ovakvim slučajevima, kod je toliko izdeljen na nezavisne celine, da ne postoji očigledan izbor za korišćenje globalnih lokacija za skladištenje.

Da sumiramo - slobodno koristite obe tehnike za razmenu podataka. Savet je da koristite parametre pre nego globalne promenljive, ali postoje određeni slučajevi kada su globalni podaci mnogo pogodniji i sigurno nije greška koristiti ovu tehniku.

Funkcija Main()

Prešli smo veći deo tehnika koje se koriste u kreiranju i upotrebi funkcija. Sada ćemo se vratiti unazad da izbliza pogledamo funkciju Main().

Rekli smo za funkciju Main () da je ulazna tačka za C# aplikaciju i da izvršenje ove funkcije prati izvršenje same aplikacije. Takođe smo videli da ova funkcija ima parametar string[ ] args, ali još nismo videli šta taj parametar predstavlja. Videćemo šta je ovaj parametar i kako se upotrebljava.

Upamtite da postoje četiri različita potpisa koja možemo koristiti za funkciju Main():

- static void Main(),
- static void Main(string[] args),
- static int Main(),
- static int Main(string [] args).

Možemo izostaviti argument args koji je ovde naveden. Verziju sa argumentom korišten jer se ona automatski pravi kada pravimo konzolnu aplikaciju u VS-u.

Treća i četvrta verzija vraćaju vrednost tipa int, koju možemo koristiti da označimo kako će se završiti aplikacija, a često se koristi da bi ukazala na grešku. Uopšteno, vraćena vrednost 0 označava da se program normalno završio (odnosno, da se aplikacija izvršila i da se može zaustaviti na siguran način).

Parametar args funkcije Main() metod je za prihvatanje informacija izvan aplikacije, navedenih u toku izvršenja programa. Ova informacija uzima oblik parametara komandne linije.

Kada izvršavamo aplikaciju preko komandne linije, često smo u mogućnosti da direktno navedemo informaciju kao što je datoteka koja se učitava u izvršenju aplikacije. Na primer, pogledajte aplikaciju Notepad u Windowsu. Možemo je pokrenuti tako što upišemo notepad u komandnu liniju, ili u okviru prozora koji se pojavljuje kada odaberemo opciju **Run** iz Windows menija Start. Takođe, možemo uneti nešto nalik na notepad "myfile.txt" na ovim lokacijama. Rezultat će biti to da će Notepad učitati datoteku "myfile.txt" kada se pokrene, ili će ponuditi da napravi ovu datoteku. Ovde je "myfile.txt" argument komandne linije. Možemo napisati konzolnu aplikaciju koja radi na sličan način koristeći parametar args.

Kada se konzolna aplikacija pokrene, svaki parametar komandne linije koji je naveden smešta se u niz args. Zatim te parametre možemo koristiti u aplikaciji.

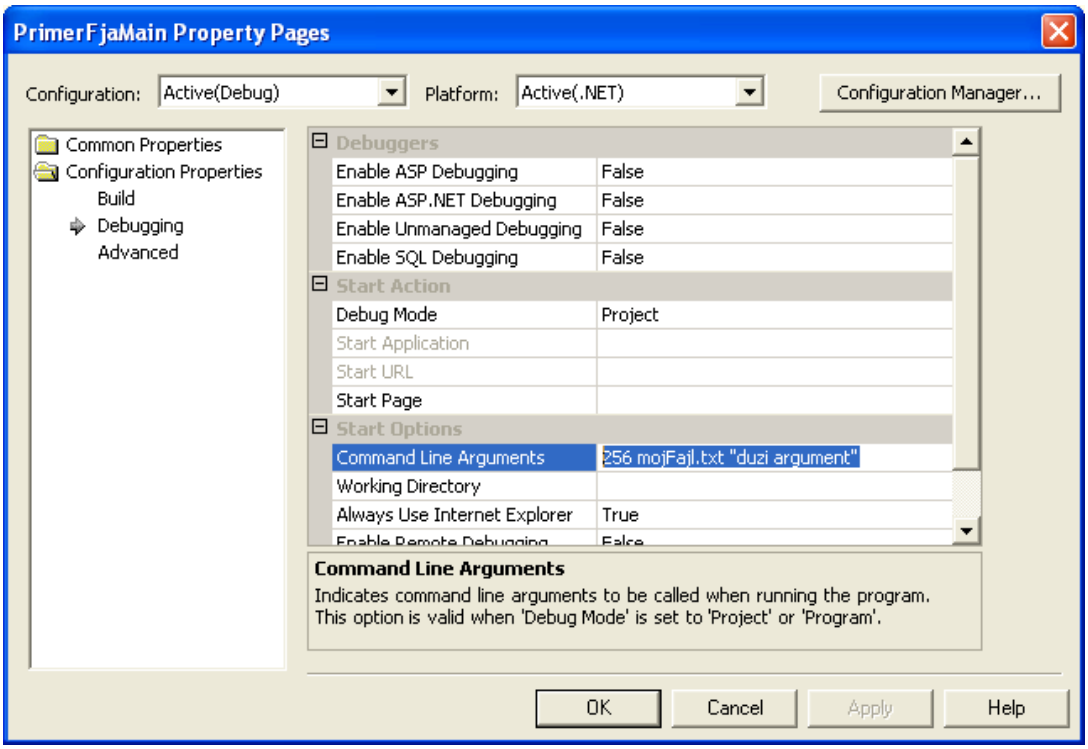
Pogledajmo praktični primer toga.

Vežba br. 16.

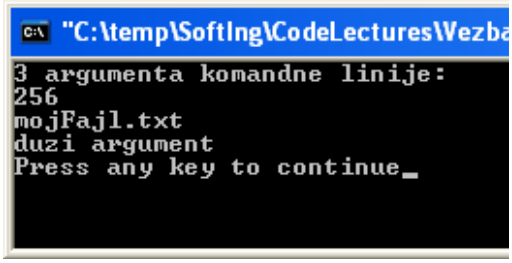
Napravite novi projekat konzolne aplikacije birajući File | New | Project... iz menija: Izaberite Visual C# Projects direktorijum unutar prozora Project Types: i tip projekta Console Application u okviru prozora Templates: (za ovo ćete morati malo da pomerite prozor na dole). U okviru za tekst Location; promenite putanju u **C:\Temp\SoftIng\LecturesCode\Vezba16** (ovaj će direktorijum biti automatski napravljen ukoliko već ne postoji), i ostavite podrazumevani tekst u okviru za tekst Name: **PrimerFjaMain.** Dodajte sledeći kod u Class1.cs:

```
8   class Class1
9   {
10    /// <summary>
11    /// The main entry point for the application.
12    /// </summary>
13    [STAThread]
14    static void Main(string[] args)
15    {
16      Console.WriteLine("{0} argumenta komandne linije:", args.Length);
17      foreach (string arg in args)
18        Console.WriteLine(arg);
19    }
20  }
21 }
22
```

Otvorite stranicu sa svojstvima projekta (pritisnite ime projekta **PrimerFjaMain** desnim tasterom miša u prozora Solution Explorer i izaberite Properties).
Izaberite stranicu Configuration Properties I Debbuging i dodajte argumente komandne linije koje želite u polje Command Line Arguments:



Pokrenite aplikaciju.



Kako to radi

Kod koji se koristi ovde vrlo je jednostavan:

```
Console.WriteLine("{0} argumenta komandne linije:", args.Length);
foreach (string arg in args)
  Console.WriteLine(arg);
```

Parametar args koristimo kao bilo koji drugi niz tipa string - samo ispisujemo na ekran šta je definisano.

U ovom primeru obezbedili smo argumente kroz dijalog Project Properties u VS-u. To je od pomoći kada se koriste isti argumenti komandne linije pri startovanju aplikacije iz VS-a, umesto da se svaki put unose u komandnu liniju. Isti rezultat bi se postigao kada bismo otvorili komandnu liniju u istom direktorijumu u kome se nalazi izlaz projekta (**C:\Temp\SoftIng\LecturesCode\Vezba16\bin\Debug**), i zatim upišemo sledeće:

256 mojFajl.txt "duzi argument"

Upamtite da je svaki argument od narednog odvojen razmakom, ali argumente možemo navesti i u navodnicima ako želimo duži argument (to je neophodno ako argument uključuje razmake, da se ne bi interpretirao kao vise argumenata).

Funkcije članice struktura

Tip struct služi za smeštanje različitih elemenata podataka na jedno mesto. Strukture su sposobne i za mnogo vise od toga. Važna dodatna sposobnost koju nude jeste mogućnost da sadrže funkcije isto kao i podatke. Ovo na prvi pogled može delovati čudno, ali je u stvari izuzetno korisno.
Kao jednostavan primer, razmotrite sledeću strukturu:

```
struct ImeKupac
{
   public string Ime, Prezime;
}
```

Ako imamo promenljivu tipa **ImeKupac**, a želimo da ispišemo njeno puno ime na konzoli, moraćemo da napravimo ime iz sastavnih delova. Možemo koristiti sledeću sintaksu za promenljivu tipa **ImeKupac** sa imenom **kupac**, na primer:

```
ImeKupac mojKupac;
mojKupac.Ime =  "Petar";
mojKupac.Prezime =  "Petrović";
```

```
Console.WriteLine("(0}  {1}  ", kupac.Ime, kupac.Prezime);
```

Dodajući funkciju strukturi, ovo možemo pojednostaviti tako što ćemo centralizovati obradu standardnih zadataka kao što je ovaj. Toj strukturi možemo dodati odgovarajuću funkciju na sledeći način:

```
struct ImeKupac
    {
      public string Ime, Prezime;
      public string ImePrezime()
      {
         return Ime + " " + Prezime;
      }
    }
```

Ovo liči na bilo koju funkciju koju smo do sada videli, osim što nije korišćen modifikator static. Treba znati da ova ključna reč nije potrebna za funkcije članica struktura. Tu funkciju možemo koristiti na sledeći način:

```
ImeKupac mojKupac;
mojKupac.Ime = "Petar";
mojKupac.Prezime = "Petrović";
Console.WriteLine (kupac.ImePrezime());
```

Sintaksa je mnogo jednostavnija i razumljivija u odnosu na prethodnu.
Važno je naglasiti da funkcija **ImePrezime()** ima direktan pristup članovima strukture **Ime** i **Prezime**. Oni se unutar strukture **ImeKupac** mogu smatrati globalnim.

Preopterećenje funkcija

Ranije u ovom predavanju smo videli da moramo složiti potpis funkcije kada je pozivamo. To znači da je potrebno imati dve funkcije koje operišu nad različitim tipovima promenljivih. Preopterećenje operatora prepoznaje ovu situaciju i omogućuje da napravimo vise različitih funkcija sa istim imenom, koje rade sa različitim tipovima parametara.

Na primer, ranije smo koristili kod koji je sadržao funkciju pod imenom MaxVrednost():

```
8   class Class1
9   {
10    [STAThread]
11    static int MaxVrednost(int[] intArray)
12    {
13      int maxProm = intArray[0];
14      for (int i = 1; i < intArray.Length; i++)
15      {
16        if (intArray[i] > maxProm)
17          maxProm = intArray[i];
18      }
19      return maxProm;
20    }
21    static void Main(string[] args)
22    {
23      int[] mojArray = {1,8,3,6,2,5,9,3,0,2};
24      int maxProm = MaxVrednost(mojArray);
25      Console.WriteLine("Maksimalna vrednost u nizu je {0}", maxProm);
26    }
27  }
28 }
29
```

Ova funkcija se može koristiti samo za niz vrednosti tipa int. Možemo obezbediti funkcije sa različitim imenima za različite tipove parametara, na primer promenom imena gornje funkcije u recimo, **IntArrayMaxVrednost**(), kao i dodavanjem funkcije kao što je **DoubleArrayMaxVrednost**() da radi sa drugim tipovima. Umesto toga, možemo samo dodati sledeću funkciju u naš kod:

```
11  static int MaxVrednost(duble[] doubleArray)
12  {
13    double maxProm = doubleArray[0];
14    for (int i = 1; i < doubleArray.Length; i++)
15    {
16      if (doubleArray[i] > maxProm)
17        maxProm = doubleArray[i];
18    }
19    return maxProm;
20  }
```

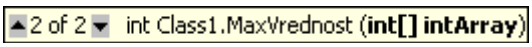
Razlika je u tome što ovde koristimo vrednosti tipa double. Ime funkcije MaxVrednost() je isto, ali je njen potpis drugačiji. Bila bi greška definisati dve funkcije sa istim imenom i potpisom, ali pošto ove funkcije imaju različit potpis, sve je u redu. Sada imamo dve verzije MaxVrednost() koje prihvataju nizove tipa int i double, i u odnosu na to vraćaju maksimalnu vrednost tipa int ili double.

Prednost je u tome što ne moramo eksplicitno navoditi koju od ove dve funkcije želimo da koristimo. Jednostavno obezbedimo niz parametara i prava funkcija će se izvršiti u zavisnosti od tipa parametra koji koristimo.

Vredi zapaziti još jednu osobinu tehnologije intelisens u VS-u. Ako u aplikaciji imamo dve malopre prikazane funkcije i krenemo da unosimo ime funkcije u (recimo) funkciji Main () , VS će prikazati moguća preopterećena imena za tu funkciju. Ako unesemo sledeće:

```
double result = MaxVrednost(
```

VS nam daje informacije o obe verzije funkcije MaxVrednost(), koje možemo birati koristeći tastere sa strelicama gore ili dole.



Kada preopterećujemo funkciju, svi aspekti potpisa funkcije biće uključeni. Možemo imati dve različite funkcije koje uzimaju parametre po vrednosti ili po referenci:

```
static void showDouble(ref int val)
{
   .........
   .........
}

static void showDouble(int val)
{
   .........
```

```
    .....
}
```

Izbor - koju od ove dve verzije ćemo koristiti, zasnovan je na tome da li poziv funkcije sadrži ključnu reč ref. Sledeće bi pozvalo verziju sa prenosom po referenci:

```
showDouble (refval) ;
```

Sledeće bi pozvalo verziju sa prenosom po vrednosti:

```
showDouble (val)
```

Alternativno, možemo imati funkcije koje se razlikuju po broju parametara koje zahtevaju i tako dalje.

Delegati

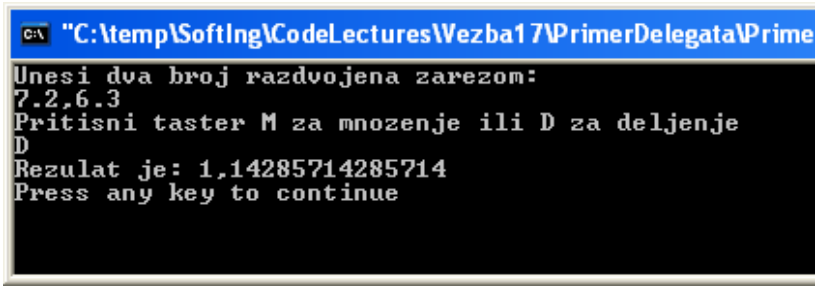
Delegat je tip koji omogućuje čuvanje referenci u funkcijama. Iako ovo zvuči vrlo stručno, mehanizam je vrlo jednostavan. Najvažniju namenu delegata razjasnićemo kada dođemo do dela knjige koji se bavi događajima i njihovom obradom. Kada kasnije budemo koristili delegate biće nam poznati, što će omogućiti razumevanje nekih komplikovanijih tema.
Delegati se deklarišu slično kao i funkcije, ali bez tela funkcije i uz upotrebu ključne red delegate. Deklaracija delegata definiše potpis funkcije koji sadrži povratni tip i listu parametara. Nakon definisanja delegata, možemo deklarisati promenljivu sa tipom tog delegata. Zatim možemo inicijalizovati promenljivu tako da sadrži referencu na bilo koju funkciju koja ima isti potpis kao i delegat. Kada to uradimo, možemo pozvati funkciju koristeći promenljivu delegata kao da je funkcija.
Kada imamo promenljivu koja pokazuje na funkciju, možemo izvesti druge operacije koje bi bilo nemoguće izvesti nekom drugom metodom. Na primer, možemo proslediti delegat nekoj funkciji kao parametar, zatim ta funkcija može pomoću delegata pozvati funkciju na koju on ukazuje, ne znajući koja će to funkcija biti sve do izvršenja.

Vežba br. 17.

Napravite novi projekat konzolne aplikacije birajući File | New | Project... iz menija:
Izaberite Visual C# Projects direktorijum unutar prozora Project Types: i tip projekta Console Application u okviru prozora Templates: (za ovo ćete morati malo da pomerite prozor na dole). U okviru za tekst Location; promenite putanju u **C:\Temp\SoftIng\LecturesCode\Vezba17** (ovaj će direktorijum biti automatski napravljen ukoliko već ne postoji), i ostavite podrazumevani tekst u okviru za tekst Name: **PrimerDelegata.** Dodajte sledeći kod u Class1.cs:

```
class Class1
{
  delegate double procesDelegati(double param1, double param2);
  [STAThread]
  static double Pomnozi(double param1, double param2)
  {
    return param1 * param2;
  }
  static double Podeli(double param1, double param2)
  {
    return param1 / param2;
  }

  static void Main(string[] args)
  {
    procesDelegati process;
    Console.WriteLine("Unesi dva broj razdvojena zarezom:");
    string ulaz = Console.ReadLine();
    int pozZareza = ulaz.IndexOf(',');
    double param1 = Convert.ToDouble(ulaz.Substring(0,pozZareza));
    double param2 = Convert.ToDouble(ulaz.Substring(pozZareza + 1,
                    ulaz.Length-pozZareza -1));
    Console.WriteLine("Pritisni taster M za mnozenje ili D za deljenje");
    ulaz = Console.ReadLine();
    if (ulaz =="M")
    {
      process = new procesDelegati(Pomnozi);
      Console.WriteLine("Rezulat je: {0}", process(param1, param2));
    }
    else
      if (ulaz =="D")
    {
      process = new procesDelegati(Podeli);
      Console.WriteLine("Rezulat je: {0}", process(param1, param2));
    }
    else
      Console.WriteLine("Niste dobro izabrali operaciju, pokrenite ponovo program !");
  }
}
}
```

Pokrenite program.



Kako to radi

Ovaj kod definiše delegat (**procesDelegati**) čiji se potpis slaže sa dve funkcije (**Pomnozi**() i **Podeli**()). Definicija delegata je sledeća:

delegate double procesDelegati(double param1, double param1);

Ključna reč delegate ukazuje na to da se definicija odnosi na delegata, a ne na funkciju (definicija se nalazi na istom mestu gde bi stajala i definicija funkcije). Zatim imamo potpis koji definiše povratnu vrednost tipa double, i dva parametra tipa double. Stvarna imena koja se koriste nisu bitna, tako da možemo nazvati tip delegata i ime parametra kako god želimo. Ovde smo delegat nazvali **procesDelegati**, a parametre **param1** i **param2**.

Kod u funkciji Main() počinje deklaracijom promenljive koja koristi novi tip delegata:

```
static void Main(string[] args)
{
  procesDelegati process;
```

Zatim imamo standardan C# kod koji traži dva broja odvojena zarezima i smešta ove brojeve u dve promenljive tipa double:

```
Console.WriteLine("Unesi dva broj razdvojena zarezom:");
string ulaz = Console.ReadLine();
int pozZareza = ulaz.IndexOf(',');
double param1 = Convert.ToDouble(ulaz.Substring(0,pozZareza));
double param2 = Convert.ToDouble(ulaz.Substring(pozZareza + 1,
                ulaz.Length-pozZareza -1));
```

Primetićete daje ovde zbog demonstracije izostavljena provera ispravnosti unosa korisnika. Da se radi o „pravom" kodu, proveli bismo još vremena proveravajući da lije korisnik uneo vrednosti u okvirima potrebnim za normalno izvršenje programa.

Zatim pitamo korisnika da li želi da se ovi brojevi pomnože ili podele:

```
Console.WriteLine("Pritisni taster M za mnozenje ili D za deljenje");
ulaz = Console.ReadLine();
```

Na osnovu korisnikov izbora, inicijalizujemo promenljivu delegata **process**.

```
if (ulaz =="M")
{
  process = new procesDelegati(Pomnozi);
  Console.WriteLine("Rezulat je: {0}", process(param1, param2));
}
else
  if (ulaz =="D")
{
  process = new procesDelegati(Podeli);
  Console.WriteLine("Rezulat je: {0}", process(param1, param2));
}
```

Da bismo dodelili referencu na funkciju promenljivoj delegata, koristimo malo čudnu sintaksu. Kao kod dodeljivanja vrednosti nizova, moramo koristiti ključnu reč **new** da napravimo novog delegata.

Posle te ključne reči navodimo tip delegata i obezbeđujemo parametar koji se odnosi na funkciju koju želimo da koristimo, uglavnom **Console.WriteLine().** Upamtite da se ovaj parametar ne slaže sa parametrom tipa delegata ili ciljnom funkcijom - ovo je jedinstvena sintaksa pri dodeljivanju delegata. Parametar je jednostavno ime funkcije koja se koristi, bez ikakvih zagrada.

Na kraju pozivamo izabranu funkciju koristeći delegat. Ovde ista sintaksa radi posao bez obzira na koju se funkciju odnosi delegat:

```
Console.WriteLine("Rezulat je: {0}", process(param1, param2));
```

Naravno unosimo i kod jedne i kod druge pitalice.
Na kraju obezbeđujemo se kad korisnik nije izabrao ni jednu od ponuđenih opcija.

```
        else
          Console.WriteLine("Niste dobro izabrali operaciju, pokrenite ponovo program !");
      }
    }
  }
```

U programu, se promenljiva delegata tretira kao ime funkcije. Za razliku od funkcija, sa ovom promenljivom se mogu izvesti dodatne operacije, kao što je njeno prosleđivanje drugoj funkciji preko parametara. Jednostavan primer ovakve funkcije bio bi:

```
static void executeFunction(procesDelegati process)
{
   process(2.2,   3.3);
}
```

To znači da ponašanje funkcije možemo kontrolisati prosleđujući joj delegate, nalik na korišćenje dopunskih modula. Na primer, možemo imati funkciju koja sortira niz tipa string po azbučnom redu.

Postoji nekoliko metoda sortiranja lista sa različitim performansama, u zavisnosti od karakteristika lista koje se sortiraju. Korišćenjem delegata možemo navesti metod koji će se koristiti, prosleđujući delegat koji pokazuje na funkciju sa algoritmom za sortiranje, funkciji koja obavlja sortiranje.

Postoji mnogo ovakvih primena delegata, ali je njihova najbolja primena kod obrade događaja.

Sažetak

U ovom predavanju smo videli kompletan pregled korišćenja funkcija u C# kodu. Mnoge od dodatnih osobina koje funkcije nude (posebno delegati) više su apstraktne i o njima ćemo govoriti tek u svetlu objektno orijentisanog programiranja, o čemu će uskoro biti reči.
Sažetak svega što smo prešli u ovom predavanju:

- Definisanje i korišćenje funkcija u konzolnim aplikacijama
- Razmena podataka sa funkcijama kroz povratne vrednosti i parametre
- Nizovi parametara
- Prosleđivanje vrednosti preko referenci ili preko vrednosti
- Navođenje izlaznih parametara za dodatne povratne vrednosti
- Koncept opsega važenja promenljivih
- Detalji o funkciji Main(), uključujući upotrebu parametra komandne linije
- Korišćenje funkcija u strukturama
- Preopterećenje funkcija
- Delegati

Vežbe

1. Sledeće dve funkcije imaju greške. Koje su to greške?

```
static bool Write()
{
   Console.WriteLine(”Izlazni tekst funkcije.”);
}

static void mojaFunkcija(string label, params int[] args, bool prikaziLabel)
{
   if (prikaziLabel)
        Console.WriteLine(label);
   foreach (int i in args)
        Console.WriteLine(”{0} ”, i)
}
```

2. Napišite aplikaciju koja koristi dva argumenta komandne linije, tako da smesti vrednosti u promenljive tipa string ili tipa int. Zatim prikažite ove vrednosti.

3. Napravite delegat i koristite ga da oponaša funkciju Console. ReadLine() kada se traži korisnikov unos.

4. Prepravite sledeću strukturu tako da uključuje funkciju koja vraća ukupnu cenu za poradžbinu:

```
struct porudzbina
{
   public int brojPorudzbine
   public string imeArtikla
   public double vredPorudzbine
}
```

5. Dodajte još jednu funkciju strukturi **porudzbina** koja vraća formatiran string na sledeći način: svi unosi koji su u kurzivu i uokvireni u ugaone zagrade moraju se zameniti odgovarajućim vrednostima.

Događaji

Ovo je poslednje poglavlje OOP sekcije ove knjige, u kojem opisujemo jednu od najčešće korišćenih OOP tehnika u .NET-u: **događaje**.

Kao i uvek, počećemo od osnovnih stvari i prodiskutovati o tome šta su to zaista događaji. Posle toga, pogledaćemo primere jednostavnih događaja i šta se sve s njima može uraditi. Kada završimo sa tim, videćemo kako da kreiramo i koristimo sopstvene događaje.

Dakle, pogledajmo šta su to događaji.

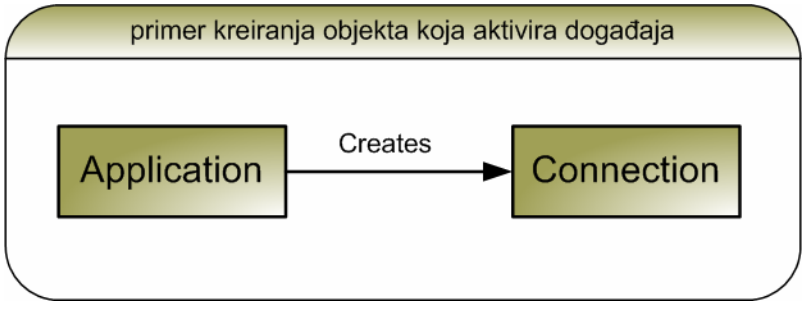
Šta je događaj?

Događaji su slični izuzecima u tome što su aktivirani od strane objekata, a mi možemo obezbediti kod koji reaguje na njih. Međutim, postoje i neke razlike. Jedna od najvažnijih jeste ta da nema ekvivalentne try. . . catch strukture za obradu događaja. Umesto toga, moramo se prijaviti na događaj. Prijavljivanje na događaj podrazumeva pisanje koda koji će se izvršiti kada je događaj aktiviran, u obliku rutine za obradu događaja.

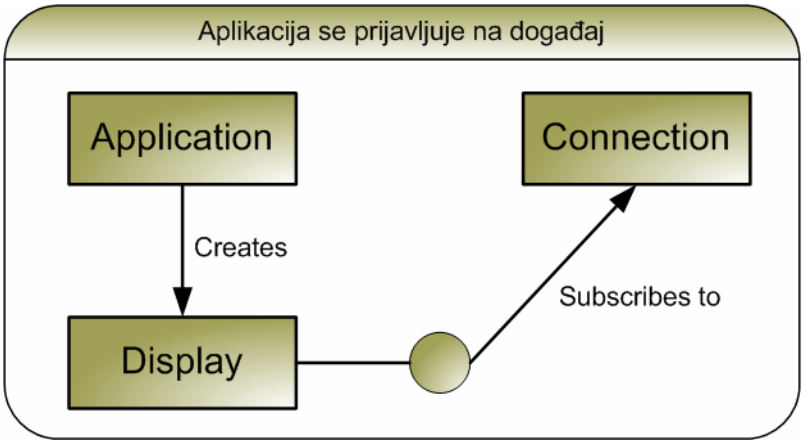
Događaj može imati vise ratina za obradu događaja, koje će sve biti pozvane kada je događaj aktiviran. Rutine za obradu događaja mogu biti deo iste klase kojoj pripada objekat koji aktivira događaj, a mogu pripadati i nekim drugim klasama.
Rutine za obradu događaja su jednostavne funkcije. Jedina restrikcija za ove funkcije jeste ta da se potpis (tip podatka koji se vraća, i parametri) mora podudarati sa onim što zahteva događaj. Ovaj potpis je deo definicije događaja, određen od strane delegata.

Upravo zato što se koristi u događajima, delegat je moćno oružje. To je razlog što smo im posvetili nešto više vremena predavanju o funkcijama. Možda biste hteli ponovo da pročitate ovu sekciju, da se podsetite šta je to delegat i kako se koristi.

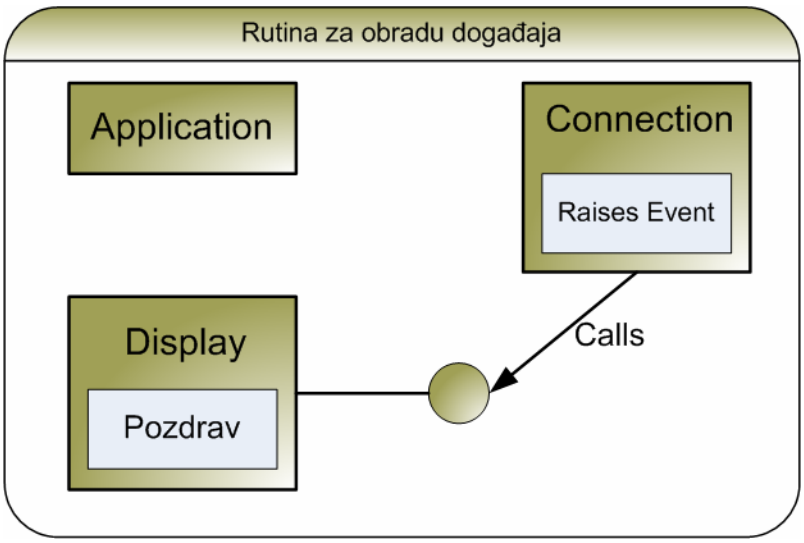
Sekvenca obrade izgleda ovako:
Prvo - aplikacija kreira objekat koji može aktivirati događaj. Na primer, neka je aplikacija zadužena za instant poruke, i objekat koji ona kreira predstavlja vezu sa udaljenim korisnikom. Objekat veze može aktivirati događaj, npr. kada stigne poruka od udaljenog korisnika.



Sledeće - aplikacija se prijavljuje na događaj. Ovo bismo mogli postići definisanjem funkcije koja bi se koristila sa delegat tipom koji je odredio događaj, i koja bi imala parametar - referencu na događaj. Rutina za obradu događaja može biti metoda na nekom drugom objektu. Neka to bude, npr. objekat koji predstavlja uređaj za prikaz instant poruke.



Kada je događaj aktiviran, aplikacija je obaveštena. Kada stigne instant poruka kroz objekat veze, bide pozvana rutina za obradu događaja na uređaju za prikaz. Pošto koristimo standardnu metodu, objekat koji aktivira događaj može proslediti, kroz parametre, bilo koju relevantnu informaciju praveći tako raznovrsne događaje. U našem slučaju, jedan parametar može biti tekst iz instant poruke. Ovaj tekst može se prikazati pomoću ratine za obradu događaja na uređaju za prikaz.



Korišćenje događaja

U ovoj sekciji pogledaćemo kod ratine za obradu događaja, a zatim ćemo videti kako možemo definisati i koristiti naš sopstveni kod.

Obrada događaja

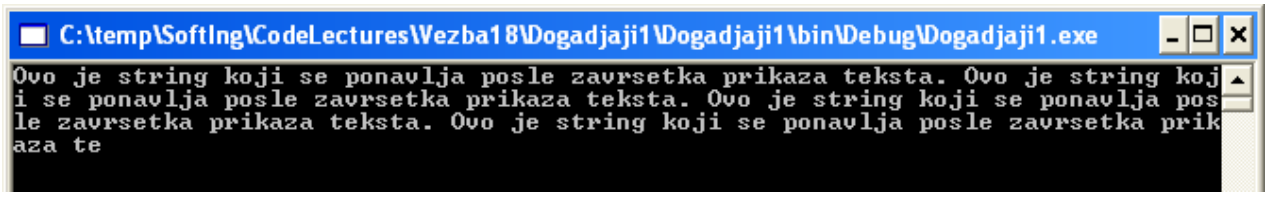
Kao što smo već rekli, za obradu događaja neophodno je prijaviti događaj, što znači obezbediti ratinu za obradu događaja čiji se potpis podudara sa delegatom koji koristi događaj. Pogledajmo primer koji koristi jednostavan objekat Timer koji aktivira događaj, posle čega će biti pozvana rutina za obradu događaja.

Vežba br. 18.

Napravite novi projekat konzolne aplikacije birajući File | New | Project... iz menija:

Izaberite Visual C# Projects direktorijum unutar prozora Project Types:, i tip projekta Console Application u okviru prozora Templates: (za ovo ćete morati malo da pomerite prozor na dole). U okviru za tekst Location: promenite putanju u **C:\Temp\SoftIng\LecturesCode\Vezba18** (ovaj će direktorijum biti automatski napravljen ukoliko već ne postoji), i ostavite podrazumevani tekst u okviru za tekst Name: **Dogadjaji1** Dodajte sledeći kod u Class1.cs:

```
using System;
using System.Timers;

namespace Dogadjaji1
{
  /// <summary>
  /// Summary description for Class1.
  /// </summary>
  class Class1
  {

    static int brojac = 0;

    static string displayString =
      "Ovo je string koji se ponavlja posle završetka prikaza teksta.  A prekida posle 100 milsek. ";

    [STAThread]
    static void Main(string[] args)
    {
      Timer mojTimer = new Timer(100);
      mojTimer.Elapsed += new ElapsedEventHandler(WriteChar);
      mojTimer.Start();
      Console.ReadLine();
    }

    static void WriteChar(object source, ElapsedEventArgs e)
    {
      Console.Write(displayString[brojac++ % displayString.Length]);
    }
  }
}
```

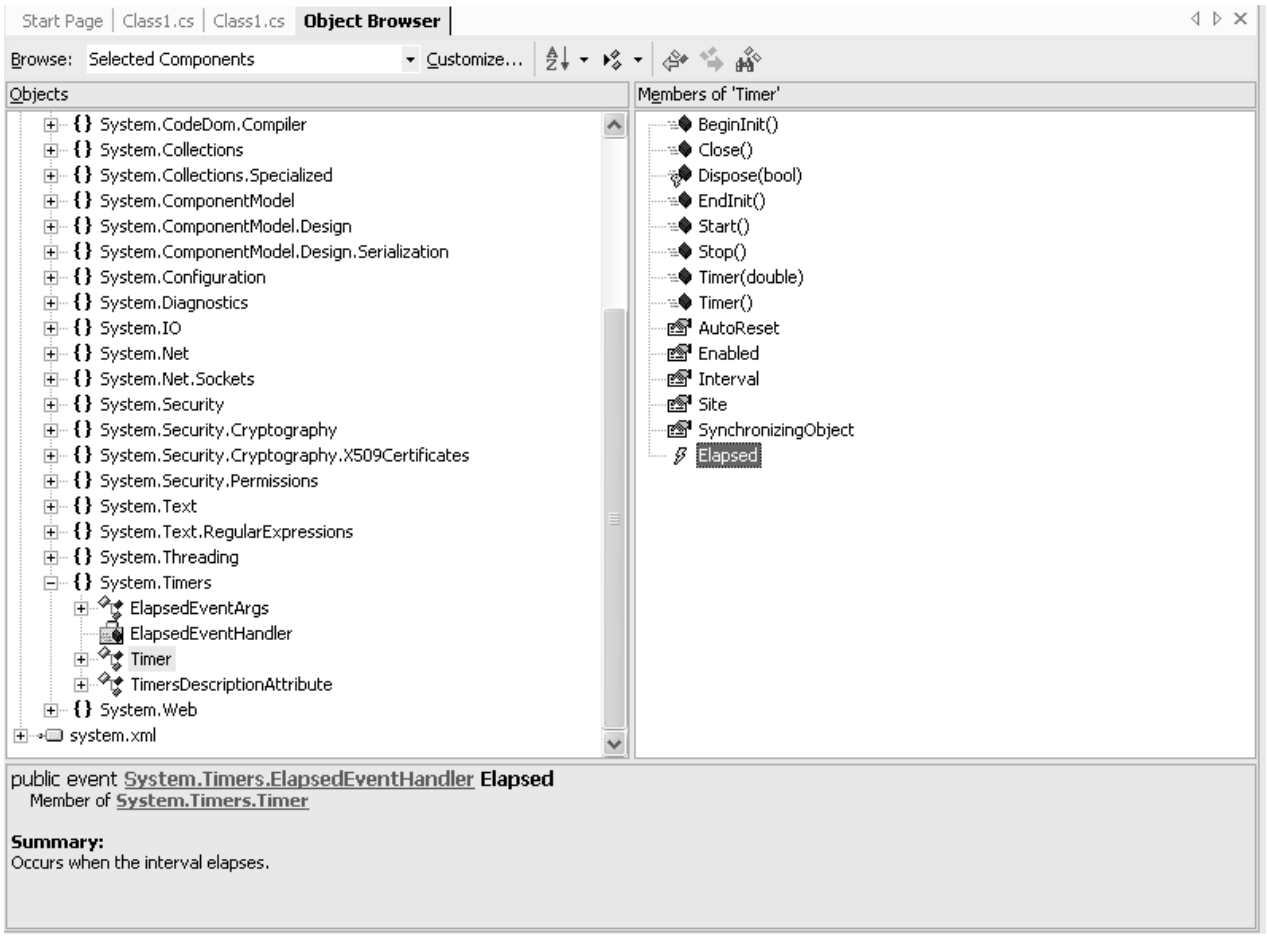


Kako to radi

Objekat koji koristimo za aktiviranje događaja je instanca klase **System.Timers.Timer**. Ovaj objekat je inicijalizovan vremenskim periodom (u milisekundama). Kada se objekat Timer startuje korišćenjem metode Start(), aktiviran je niz događaja razdvojenih vremenskim periodom koji je određen. Funkcija Main() inicijalizuje objekat Timer vremenskim periodom od 100 milisekundi, tako da će aktivirati 10 događaja u sekundi:

```
22    static void Main(string[] args)
23    {
24      Timer mojTimer = new Timer(100);
```

Objakat Timer ima događaj koji se zove Elipsed. Možemo ga videti ako koristimo prozor Object Broser:



Potpis rutine za obradu ovog događaja definisan je u tipu delegata System.Timers.ElapsedEventHandler, koji je jedan od standardnih delegata definisanih u .NET okruženju. Ovaj delegat se koristi za funkcije koje se podudaraju sa sledećim potpisom:

void functionName(object source, ElapsedEventArgs e);

Objekat Timer šalje referencu na samog sebe, u prvom parametra, i instancu ElapsedEventArgs klase u drugom parametru. Za sada možemo ignorisati ove parametre, ali pogledaćemo ih kasnije.

U kodu imamo funkciju koja se podudara sa ovim potpisom:

```
31    static void WriteChar(object source, ElapsedEventArgs e)
32    {
33      Console.Write(displayString[brojac++ % displayString.Length]);
34    }
35  }
36 }
```

Ova metoda koristi dva statička polja klase Class1, **brojac** i displayString, za prikaz jednog znaka. Za svaki sledeći poziv metode, znak koji se prikazuje biće različit.

Poslednje što moramo obaviti jeste da povežemo događaj i rutina za obradu događaja - da ga prijavimo. To postižemo operatorom += dodajući događaj rutinu za obradu događaja, u formi nove instance delegate inicijalizovane našom ratinom za obradu događaja:

```
21  [STAThread]
22  static void Main(string[] args)
23  {
24   Timer mojTimer = new Timer(100);
25
26   mojTimer.Elapsed += new ElapsedEventHandler(WriteChar);
```

Ova komanda (koja koristi malo čudnu sintaksu, svojstvenu delegatima) dodaje rutinu za obradu događaja u listu koja će biti pozvana kada se aktivira događaj Elapsed. Možemo dodati koliko god hoćemo rutina za obradu događaja u ovu listu, dokle god one zadovoljavaju postavljene kriterijume. Sve ratine za obradu događaja će biti pozvane kada se događaj aktivira.

Sve što ostaje funkciji Main() jeste da pokrene Timer:

```
27
28      mojTimer.Start();
```

Pošto ne želimo da okončamo aplikaciju pre nego što obradimo bilo koji događaj, stavljamo funkciju Main() na čekanje. Najlakši način da ovo postignemo jeste kroz zahtev za unos podataka od strane korisnika, jer se obrada neće završiti dok korisnik ne unese red teksta i/ili pritisne Enter.

```
29    Console.ReadLine();
30  }
```

Iako se obrada u funkciji Main() završava ovde, obrada u objektu Timer se nastavlja. Kada ovaj objekat aktivira događaje, poziva se metoda WriteChar() koja se istovremeno izvršava sa Console.ReadLine() iskazom.

Definisanje događaja

Pogledajmo kako da definišemo i koristimo naše sopstvene događaje. Implementiraćemo verziju primera za instant poruke sa početka ovog poglavlja. Kreiraćemo objekat **Konekcija** koji aktivira događaje obrađene od strane objekta **Display**.

Primer definisanja događaja

Vežba br. 19.

Napravite novi projekat konzolne aplikacije birajući File | New | Project... iz menija:

Izaberite Visual C# Projects direktorijum unutar prozora Project Types:, i tip projekta Console Application u okviru prozora Templates: (za ovo ćete morati malo da pomerite prozor na dole). U okviru za tekst Location: promenite putanju u **C:\Temp\SoftIng\LecturesCode\Vezba19** (ovaj će direktorijum biti automatski napravljen ukoliko već ne postoji), i ostavite podrazumevani tekst u okviru za tekst Name: **Dogadjaji2** Dodajte sledeći kod u Class1.cs:

Dodajte novu klasu Konekcija u Konekcija.cs:

```
1  using System;
2  using System.Timers;
3
4  namespace Dogadjaji2
5  {
6    /// <summary>
7    /// Summary description for Konekcija.
8    /// </summary>
9    ///
10   public delegate void MessageHandler(string messageText);
11
12   public class Konekcija
13   {
14
15     public event MessageHandler MessageArrived;
16
17     private Timer pollTimer;
18
19     public Konekcija()
20     {
21       pollTimer = new Timer(100);
22       pollTimer.Elapsed += new ElapsedEventHandler(ProveraPoruke);
23     }
24     public void Connect()
25     {
26       pollTimer.Start();
27     }
28
29     public void Disconnect()
30     {
31       pollTimer.Stop();
32     }
33
29     private void ProveraPoruke(object source, ElapsedEventArgs e)
30     {
31       Console.WriteLine("Provera nove Poruke.");
32       Random random = new Random();
33       if ((random.Next(9) == 0) && (MessageArrived != null))
34       {
35         MessageArrived(" #### Pozdrav MAMA ! ###");
36       }
37     }
38   }
39 }
40
```

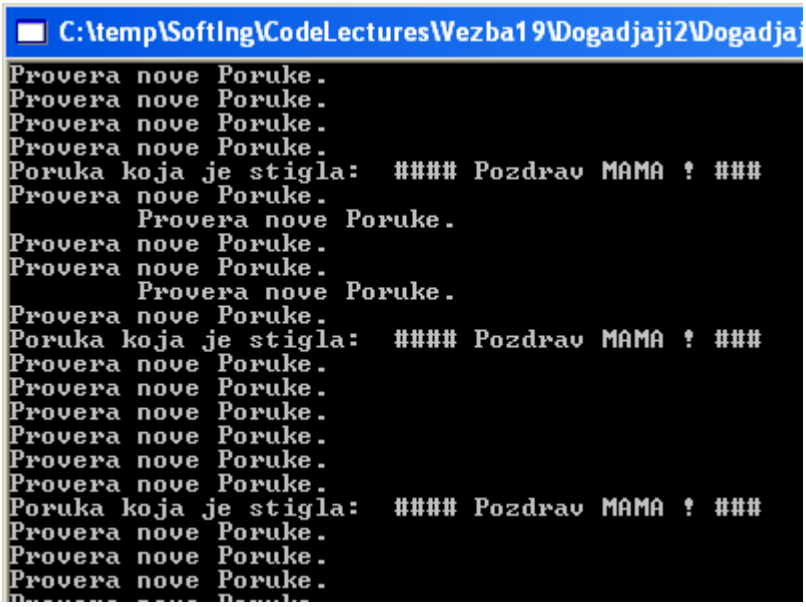
Dodajte novu klasu Display u Display.cs:

```
1  using System;
2
3  namespace Dogadjaji2
4  {
5    /// <summary>
6    /// Summary description for Display.
7    /// </summary>
8    public class Display
9    {
10
11     public void DisplayMessage(string poruka)
12     {
13       Console.WriteLine("Poruka koja je stigla: {0}", poruka);
14     }
15
16   }
17 }
18
```

Izmenite kodu Class1.cs:

```
5  using System;
6
7  namespace Dogadjaji2
8  {
9
10   class Class1
11   {
12
13     [STAThread]
14     static void Main(string[] args)
15     {
16       Konekcija mojaKonekcija = new Konekcija();
17
18       Display myDisplay = new Display();
19
20       mojaKonekcija.MessageArrived +=
21         new MessageHandler (myDisplay.DisplayMessage);
22
23       mojaKonekcija.Connect();
24       Console.ReadLine();
25     }
26   }
27 }
28
```

Pokrenite aplikaciju.



Kako to radi

Najveći posao u ovoj aplikaciji obavlja klasa **Konekcija**. Instance ove klase koriste objekte Timer, kao što smo videli u prvom primeru ovog poglavlja, inicijalizujući ih u konstruktoru klase. Takođe, obezbeđuju pristup statusu objekta Timer (omogućiti ili onemogućiti) kroz metode Connect() i Disconnect():

```
11
12  public class Konekcija
13  {
14
15
16
17    private Timer pollTimer;
18
19    public Konekcija()
20    {
21      pollTimer = new Timer(100);
22      pollTimer.Elapsed += new ElapsedEventHandler(ProveraPoruke);
23    }
24    public void Connect()
25    {
26      pollTimer.Start();
27    }
28
29    public void Disconnect()
30    {
31      pollTimer.Stop();
32    }
```

Takođe, u konstruktoru prijavljujemo rutinu za obradu događaja za **Elapsed** događaj, kao što smo to radili u prvom primeru. Metoda **CheckForMessage**() aktivira događaj, u proseku jednom od svakih deset puta kada je pozvana. Pre nego što predemo na kod za ovu metodu, pogledajmo definiciju samog događaja.

Pre nego što definišemo metodu, moramo definisati tip delegata koji se koristi sa događajem, tj. tip delegata koji određuje potpis koji mora imati i rutina za obradu događaja. To radimo uz pomoć standardne sintakse za delegate - definišemo ga kao javni tip unutar imenovanog prostora **Dogadjaji2**, da bi bio dostupan spoljašnjem kodu:

```
4  namespace Dogadjaji2
5  {
6    /// <summary>
7    /// Summary description for Konekcija.
8    /// </summary>
9    ///
10   public delegate void MessageHandler(string messageText);
11
```

Ovaj tip delegata, koji smo nazvali **MessageHandler**, jeste potpis za void funkcije koje imaju jedan string parametar. Taj parametar možemo koristiti za prenos instant poruke primljene od objekta **Konekcija** do objekta Display.
Kada je delegat definisan (ili je odgovarajući postojeći delegat lociran), možemo definisati i sam događaj kao član klase **Konekcija**:

```
12   public class Konekcija
13   {
14
15     public event MessageHandler MessageArrived;
```

Jednostavno imenujemo događaj (ovde smo koristili ime **MessageArrived**) i deklarišemo ga, koristeći ključnu reč **event** i tip delegata (tip delegate **MessageHandler** koji smo već definisali). Kada je događaj deklarisan na ovaj način, možemo ga aktivirati pozivajući ga po imenu, kao da se radi o metodu sa potpisom određenim od strane delegata. Na primer, možemo aktivirati događaj na sledeći način:

```
35        MessageArrived(" #### Pozdrav MAMA ! ###");
36      }
```

:

Da je delegat definisan bez parametara, mogli smo napisati:

MessageArrived();

Mogli smo definisati i vise od jednog parametra, koji bi zahtevali vise koda za aktiviranje događaja. Naša metoda **ProveraPoruke()** izgleda ovako:

```
33
29   private void ProveraPoruke(object source, ElapsedEventArgs e)
30   {
31     Console.WriteLine("Provera nove Poruke.");
32     Random random = new Random();
33     if ((random.Next(9) == 0) && (MessageArrived != null))
34     {
35       MessageArrived(" #### Pozdrav MAMA ! ###");
36     }
37   }
38 }
39 }
40
```

Koristimo instancu klase Random koju smo videli ranije, za slučajan izbor broja između 0 i 9, kao i za aktiviranje događaja ukoliko je izabrani broj 0, što će se desiti u 10% slučajeva. Ovo simulira ispitivanje veze radi provere da li je poruka stigla, što neće biti slučaj svaki put kada proveravamo.

Skrećemo pažnju da događaj aktiviramo samo ako je izraz **MessageArrived ! = null** tačan. Ovaj izraz, koji koristi sintaksu delegata na neuobičajen način, u stvari znači: „Da li uopšte postoji nešto što bi aktiviralo događaj?" Ako tako nešto ne postoji, MessageArrived će uzeti vrednost **null**, tako da nema nikakvog smisla aktivirati događaj.

Klasa koja će reagovati kada je događaj aktiviran jeste klasa Display. Ona sadrži jednu metodu **DisplayMessage**():

```
8   public class Display
9   {
10
11      public void DisplayMessage(string poruka)
12      {
13        Console.WriteLine("Poruka koja je stigla: {0}", poruka);
14      }
15
16    }
17  }
18
```

Ova metoda ima odgovarajući potpis (i javna je, što je obavezno za ratine za obradu događaja koji ne pripadaju istoj klasi, kao i za objekat koji aktivira događaj), tako da je možemo koristiti za obradu događaja MessageArrived.

Sve što preostaje da se uradi u funkciji **Main**() jeste inicijalizacija dve instance klasa **Konekcija** i **Display**, i njihovo povezivanje. Kod je sličan onome iz prvog primera:

```
13    [STAThread]
14    static void Main(string[] args)
15    {
16      Konekcija mojaKonekcija = new Konekcija();
17
18      Display myDisplay = new Display();
19
20      mojaKonekcija.MessageArrived +=
21        new MessageHandler (myDisplay.DisplayMessage);
22
23      mojaKonekcija.Connect();
24      Console.ReadLine();
25    }
26  }
27 }
28
```

Ponovo pozivamo metod **Console.ReadLine()** da pauzira obradu u funkciji **Main**() kada predemo na metodu **Connect()** objekta **Konekcija**.

Višenamenske rutine za obradu događaja

Potpis koji smo videli ranije za događaj **Timer.Elapsed**, sadrži dva parametra čiji se tipovi često sreću u rutinama za obradu događaja. Ti parametri su:

- **object source** - referenca na objekat koji aktivira događaj.
- **ElapsedEventArgs e** - parametri koje šalje događaj.

Razlog zbog kojeg koristimo parametar tipa **object** za ovaj događaj, kao i za mnoge druge događaje, jeste taj što ćemo često hteti da koristimo jednu rutinu za obradu događaja za vise identičnih događaja aktiviranih od strane različitih objekata. Na ovaj način možemo tačno znati koji objekat aktivira događaj.

Proširimo malo poslednji primer.

Vežba br. 20.

Napravite novi projekat konzolne aplikacije birajući File | New | Project... iz menija:

Izaberite Visual C# Projects direktorijum unutar prozora Project Types:, i tip projekta Console Application u okviru prozora Templates: (za ovo ćete morati malo da pomerite prozor na dole). U okviru za tekst Location: promenite putanju u **C:\Temp\SoftIng\LecturesCode\Vezba20** (ovaj će direktorijum biti automatski napravljen ukoliko već ne postoji), i ostavite podrazumevani tekst u okviru za tekst Name: **Dogadjaji3** Dodajte sledeći kod u Class1.cs:

Kopirajte kod iz Class1.cs, Konekcija.cs i Display.cs iz projekta **Dogadjaji2**. Vodite računa da promenite imenovani prostor iz **Dogadjaji2**u **Dogadjaji3**.

Dodajte novu klasu, **MessageArrivedEventArgs**, u datoteku **MessageArrivedEventArgs.cs:** using System; namespace **Dogadjaji3**

```
1  using System;
2
3  namespace Dogadjaji3
4  {
5    public class MessageArrivedEventArgs : EventArgs
6    {
7      private string poruka;
8
9      public string Poruka
10     {
11       get
12       {
13         return poruka;
14       }
15     }
16
17     public MessageArrivedEventArgs()
18     {
19       poruka = "Nijedna poruka nije poslata.";
20     }
21
22     public MessageArrivedEventArgs(string novaPoruka)
23     {
24       poruka = novaPoruka;
25     }
26   }
27 }
28
```

Izmenite Connection.cs:

```
using System;
using System.Timers;

namespace Dogadjaji3
{
  public delegate void MessageHandler(Konekcija source,
        MessageArrivedEventArgs e);

  public class Konekcija
  {
    public event MessageHandler MessageArrived;

    private string naziv;

    public string Naziv
    {
      get
      {
        return naziv;
      }
      set
      {
        naziv = value;
      }
    }

    private Timer pollTimer;

    public Konekcija()
    {
      pollTimer = new Timer(100);
      pollTimer.Elapsed += new ElapsedEventHandler(ProveraPoruke);
    }
    public void Connect()
    {
      pollTimer.Start();
    }

    public void Disconnect()
    {
      pollTimer.Stop();
    }

    private void ProveraPoruke(object source, ElapsedEventArgs e)
    {
      Console.WriteLine("Provera nove Poruke.");
      Random random = new Random();
      if ((random.Next(9) == 0) && (MessageArrived != null))
      {
        MessageArrived(this, new MessageArrivedEventArgs("#### Pozdrav MAMA ! ###"));
      }
    }
  }
}
```

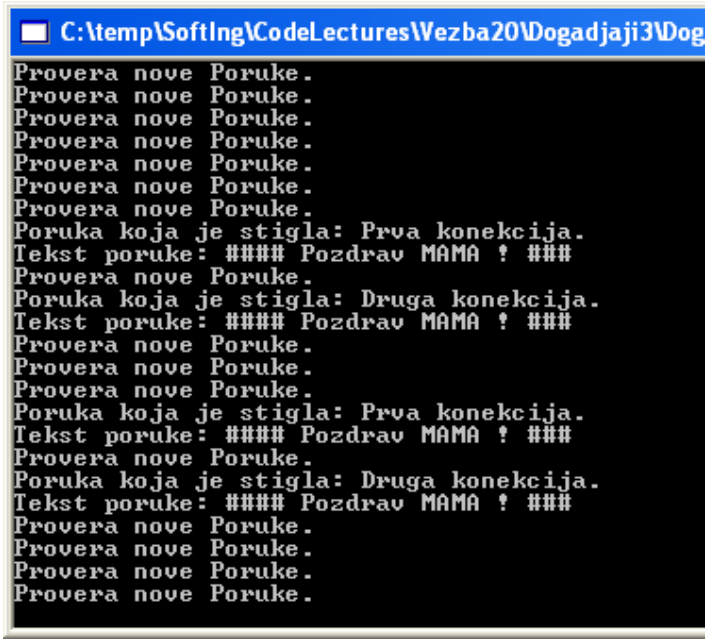
Izmenite **Display.cs**:

```
using System;

namespace Dogadjaji3
{
 /// <summary>
 /// Summary description for Display.
 /// </summary>
 public class Display
 {

  public void DisplayMessage(Konekcija source, MessageArrivedEventArgs e)
  {
   Console.WriteLine("Poruka koja je stigla: {0}", source.Naziv);
   Console.WriteLine("Tekst poruke: {0}", e.Poruka);
  }
 }
}
```

Izmenite **Class1.cs**:

```
[STAThread]
static void Main(string[] args)
{
 Konekcija mojaKonekcija1 = new Konekcija();
 mojaKonekcija1.Naziv = "Prva konekcija.";

 Konekcija mojaKonekcija2 = new Konekcija();
 mojaKonekcija2.Naziv = "Druga konekcija.";

 Display myDisplay = new Display();

 mojaKonekcija1.MessageArrived +=
  new MessageHandler(myDisplay.DisplayMessage);

 mojaKonekcija2.MessageArrived +=
  new MessageHandler(myDisplay.DisplayMessage);

 mojaKonekcija1.Connect();
 mojaKonekcija2.Connect();

 Console.ReadLine();
```

Pokrenite aplikaciju:



Kako to radi

Slanjem reference na objekat koji aktivira događaj kao jednog od parametra ratine za obradu događaja, možemo prilagoditi odgovor ratine na individualni objekat. Referenca daje pristup izvornom objektu, uključujući i njegova svojstva.
Slanjem parametara koji su sadržani u klasi izvedenoj iz **System.EventArgs** (kao što je **ElapsedEventArgs**), možemo obezbediti bilo koje dodatne informacije koje su neophodne kao parametre (npr. parametar **Poruka** naše MessageArrivedEventArgs klase).

Polimorfizam će dodatno koristiti parametrima. Mogli bismo definisati ratinu za obradu događaja **MessageArrived**:

```
10
11      public void DisplayMessage(object source, EventArgs e)
12      {
13        Console.WriteLine("Poruka koja je stigla: {0}", ((Konekcija)source).Naziv);
14        Console.WriteLine("Tekst poruke: {0}", ((MessageArrivedEventArgs)e).Poruka);
15      }
16      }
17    }
18
```

Izmenite definiciju delegata u datoteci **Kolekcija.cs**:

```
4   namespace Dogadjaji3
5   {
6     public delegate void MessageHandler(object source, EventArgs e);
7
```

Aplikacija će se izvršavati kao i pre, s tim da je funkcija **DisplayMessage()** mnogostrana (bar teoretski - dodatna implementacija bi bila neophodna za kvalitetan proizvod). Ista rutina može raditi i sa dragim događajima, kao što je događaj **Timer.Elapsed**. U tom slučaju, morali bismo izmeniti rutinu tako da parametri koji se šalju kada je događaj aktiviran budu valjano obrađeni (konverzija parametara u objekte **Konekcija** i **MessageArrivedEventArgs** na ovaj način izazvala bi izuzetak; trebalo bi koristiti operator as).

Pre nego što krenemo dalje, primetite da se događaji aktiviraju u parovima, zbog dva objekta **Konekcija**. U pitanju je način na koji klasa Random generiše slučajne brojeve. Kada se kreira objekat Random, koristi se početna vrednost. Ova početna vrednost koristi kompleksnu

jednačinu za generisanje niza pseudoslučajnih brojeva. (Računari nisu u stanju da generišu prave slučajne brojeve.) Početnu vrednost za Random objekat možemo definisati u konstruktora, ali ako to ne uradimo (kao što i nismo), trenutno vreme se koristi kao početna vrednost. Pošto formiramo dva Connection objekta u dva reda koda, koji idu jedan za dragim, postoji velika verovatnoća da će se koristiti iste početne vrednosti kada objekti odgovaraju na događaj **Timer.Elapsed.** Ako jedan objekat generiše poruku, vrlo je verovatno da će i dragi to činiti, što je posledica brzine obrade. Da bismo rešili problem, moramo drugačije podesiti početne vrednosti. Jedno rešenje, koje ovde nećemo raditi, bilo bi da se obezbedi instanca klase Random kojoj se može pristupiti iz oba objekta Konekcija. Objekti bi, u tom slučaju, koristili isti niz slučajnih brojeva, što bi bitno umanjilo šansu za prikazivanje sinhronizovanih poruka.

Povratne vrednosti i rutina za obradu događaja

Svi moduli koje smo do sada videli imali su povratni tip **void**. Moguće je obezbediti neki dragi tip, ali to dovodi do problema zato što neki događaj može imati za posledicu pozivanje nekoliko rutina za obradu događaja. Ako sve ove ratine vrate neku vrednost, postaje nejasno koju od vrednosti uzimamo.

Sistem rešava problem tako što mm dozvoljava pristup samo poslednjoj vraćenoj vrednosti. Ovo je vrednost koju vraća poslednja ratina za obradu događaja koja se odnosi na neki događaj.

Možda ovo rešenje ima upotrebnu vrednost u nekoj situaciji, ali ipak preporučujemo korišćenje povratnog tipa void za ratine za obradu događaja, kao i izbegavanje korišćenja parametara tipa **out**.

Korišćenje dijaloga u C

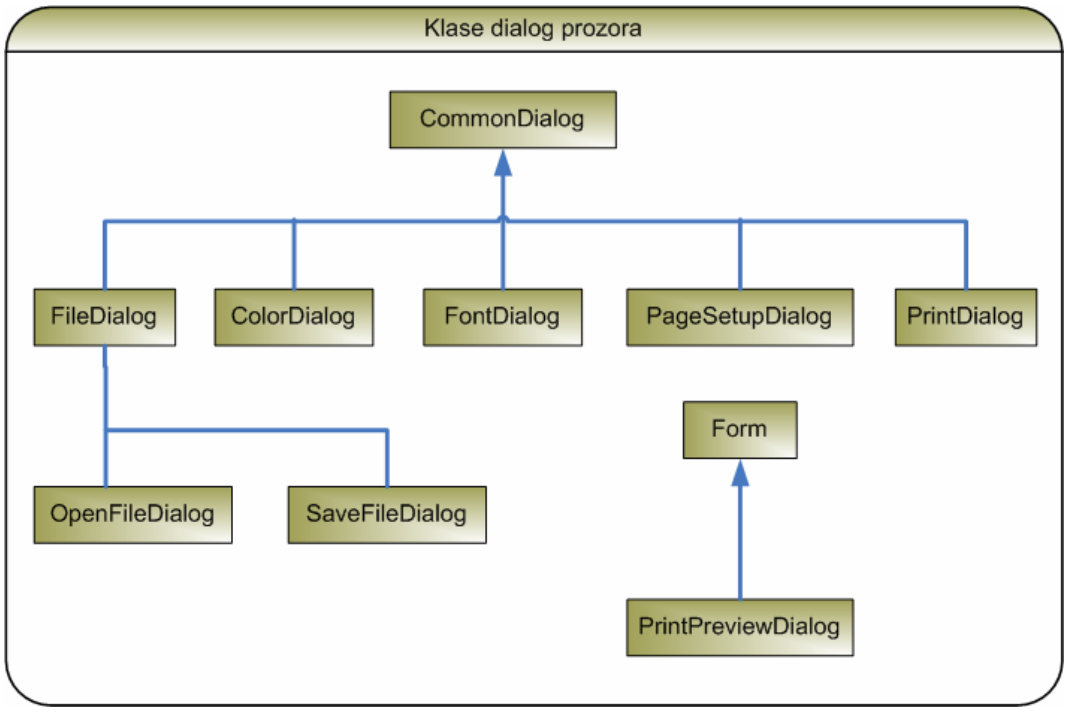
U prethodnim predavanjima videli smo kako da implementiramo menije, palete sa alatkama, SDI i MDI formulare i sl. Već znamo kako da prikažemo jednostavne okvire za poruke i da dobijemo informacije od korisnika. Napravili smo prefinjenije posebne dijaloge da upitaju korisnika za posebne informacije. Ipak, ako je moguće, trebalo bi da koristite preddefinisane dijaloge klase, pre nego posebne dijaloge za obične operacije kao što su otvaranje i snimanje datoteka ili štampanje. Ovo nije samo prednost ostvarivanja zadatka sa manje koda, već i korišćenja standardnih poznatih Windows dijaloga. .NET okruženje sadrži klase koje se kače na Windows dijaloge za otvaranje i snimanje datoteka, za pristupanje štampačima, kao i za biranje boja i fontova. Mogućnost korišćenja tih dijaloga umesto posebnih dijaloga znači da nije neophodno za korisnika da nauči kompleksnu metodologiju koja bi bila potrebna da bi se takva funkcionalnost napravila iz početka.

U ovom predavanju naučićemo kako se koriste ove klase sa standardnim dijalozima. Konkretno:

- Videćemo kako se koriste klase **OpenFileDialog** i **SaveFileDialog**.
- Naučićemo hijerarhiju klasa za štampanje u .NET-u, i pomoću klasa **PrintDialog**, **PageSetupDialog** i **PrintPreviewDialog** implementiraćemo štampanje i pregled pre štampanja.
- Upoznaćemo klase **FontDialog** i **ColorDialog** kako bismo menjali fontove i boje.

Dijalozi

Dijalog je prozor koji je prikazan u sadržini konteksta drugog prozora. Možemo da upitamo korisnika za unos podataka pre nego što nastavimo tok našeg programa. Opšti dijalog je dijalog koji se često koristi da se dobiju potrebne informacije od korisnika, i deo je Windows operativnog sistema.
Sa .NET radnim okruženjem imamo sledeće klase dijaloga:



Sve ove klase dijaloga, izuzev **PrintPreviewDialog**, izvedene su iz apstraktne klase **CommonDialog** koja ima metode da upravlja Windowsovim opštim dijalogom.

Klasa **CommonDialog** definiše sledeće metode i događaje, zajedničke za sve opšte klase dijaloga:

Metodi i događaji javne instance	Opis
ShowDialog()	Ova metoda je implementirana iz izvedene klase da prikaže opšti dijalog.
Reset()	Svaka izvedena klasa dijaloga implementira metodu Reset() da postavi sva svojstva klase dijaloga na njihove podrazumevane vrednosti.
HelpRequest	Ovaj događaj se pojavljuje kada korisnik pritisne dugme Help na opštem dijalogu.

Sve ove klase dijaloga obmotavaju opšti Windows dijalog da bi napravile dijalog dostupan za .NET aplikacije. Klasa **PrintPreviewDialog** je izuzetak, jer dodaje sopstvene elemente u Windows formular da kontroliše pregled štampanja i nije zapravo dijalog. Klase **OpenFileDialog** i **SaveFileDialog** nastale su iz apstraktne bazne klase **FileDialog** koja dodaje mogućnosti rada sa datotekama, koje su zajedničke za otvaranje i zatvaranje datoteke.
Pregledajmo gde se sve mogu koristiti različiti dijalozi:

- Da bi korisnik izabrao i pregledao datoteke za otvaranje koristi **OpenFileDialog**. Ovaj dijalog može se konfigurisati tako da se samo jedna datoteka ili višestruke datoteke mogu izabrati.
- Sa dijalogom **SaveFileDialog** korisnik može da odabere ime datoteke i potraži direktorijum u kome se jedna datoteka ili više njih mogu izabrati.
- **PrintDialog** se koristi da se izabere štampač i postave opcije za štampu.
- Da bi se konfigurisale margine strane, uglavnom se koristi **PageSetupDialog**.
- **PrintPreviewDialog** je jedan način da se uradi pregled pre štampe na ekranu sa nekoliko opcija (npr. zumiranje).
- **FontDialog** izlistava sve instalirane Windows fontove sa stilovima, veličinama i pregledom da se izabere željeni font.
- Klasa **ColorDialog** olakšava izbor boje.

Kod nekih aplikacija razvijenih od iste kompanije ne samo da nisu opšti dijalozi korišćeni, već ni stil za pravljenje posebnih dijaloga nije bio korišćen. Razvoj ovih dijaloga dao je funkcionalnost koja nije bila postojana, gde su se neka dugmad i okviri sa listom nalazili na drugim mestima, na primer OK i Cancel dugme su bili okrenuti.

Ponekad se ta nepostojanost može naći i unutar jedne aplikacije. To je frustrirajuće za korisnika i povećava vreme za završavanje zadatka.

Budite postojani u dijalozima koje pravite i koristite! Postojanost može lako da se postigne korišćenjem opštih dijaloga. Opšti dijalozi se koriste od raznih aplikacija već poznatih korisniku.

Kako se koriste dijalozi

Pošto je klasa **CommonDialog** osnovna klasa za sve klase dijaloga, tako se sve klase dijaloga slično koriste. Javne metode instance su **ShowDialog()** i **Reset()**. Metoda ShowDialog poziva

zaštićeni **RunDialog** metod instance da prikaže dijalog i konačno vraća instancu DialogResult sa informacijom o tome kakvu je korisnik interakciju vršio sa dijalogom. Metod **Reset()** zapravo postavlja svojstva klase dijaloga na njihove podrazumevane vrednosti.
Sledeći segment koda pokazuje primer kako se klasa dijaloga može koristiti. Kasnije, pogledaćemo detaljnije na svaki od ovih koraka; prvo da uvedemo koncept kako će se dijalozi koristiti.

Kao što možete videti u sledećem segmentu koda:

- Prvo se pravi nova instanca klase dijaloga.
- Zatim moramo da postavimo neka svojstva da uključe/isključe opcionalne mogućnosti i postave stanje dijaloga. U ovom slučaju, postavićemo svojstvo **Title** na **Primer**, i postavićemo fleg **ShowReadOnly** na **true**.
- Pozivajući metod **ShowDialog()**, dijalog se prikazuje, čeka i reaguje na korisnički unos.
- Ako korisnik pritisne dugme OK - dijalog se zatvara. Proveravamo pritisak na OK poredeći rezultat dijaloga sa **DialogResult**. OK. Posle toga, možemo uzeti vrednosti iz korisničkog unosa ispitujući određene vrednosti svojstava. U ovom slučaju, čuvamo vrednost svojstva **FileName** u promenljivoj **FileName**:

```
OpenFileDialog dlg = new OpenFileDialog();
dlg.Title ="Primer";
dlg.ShowReadOnly=true;

if (dlg.ShowDialog() == DialogResult.OK)
{
  string filename =dlg.FileName;
}
```

Naravno, svaki dijalog ima sopstvene opcije za podešavanje, koje tražimo u sledećim sekcijama.
Ako koristite dijalog u aplikacijama sa Windows formularima, to je čak i lakše od nekoliko redova koda. Dizajner Windows formulara pravi kod za kreiranje nove instance i postavljanje vrednosti svojstava. Mi jednostavno moramo da pozovemo metod **ShowDialog()** i dođemo do promenjenih vrednosti, kao što ćemo videti.

Dijalozi za rad sa datotekama

Dijalogom za rad sa datotekama korisnik može da odabere disk i pretražuje kroz sistem datoteka da bi izabrao datoteku. Sve što želimo da imamo od korisnika jeste ime datoteke.

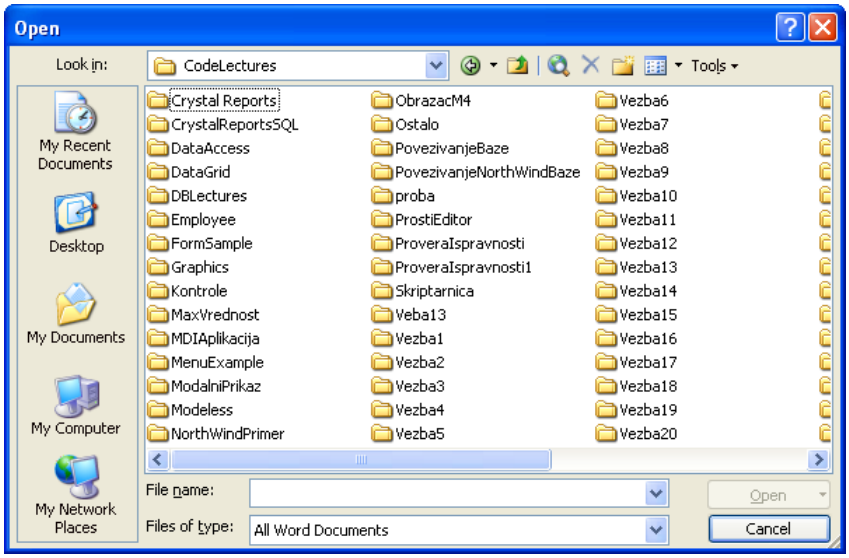
Sa dijalogom **OpenFileDialog** možemo da izaberemo ime za datoteku koju želimo da otvorimo, a korišćenjem dijaloga **SaveFileDialog** možemo navesti ime datoteke koju želimo da snimimo. Ove klase dijaloga su vrlo slične jer su izvedene iz iste apstraktne osnovne klase, iako postoje svojstva jedinstvena za svaku klasu. U ovoj sekciji prvo ćemo pogledati karakteristike klase **OpenFileDialog**, a tada ćemo pogledati gde se klasa **SaveFileDialog** razlikuje. Razvićemo primer aplikacije koja koristi i jednu i drugu klasu.

Klasa OpenFileDialog

Klasa OpenFileDialog nam omogućava da izaberemo datoteku za otvaranje. Kao što smo videli u našem primeru iznad, nova instanca klase **OpenFileDialog** se pravi pre nego što se pozove metod **ShowDialog()**.

```
OpenFileDialog dlg = new OpenFileDialog();
dlg.ShowDialog();
```

Pokretanje programa sa ova dva reda koda će rezultovati ovim dijalogom:



Kao što smo već videli, možemo postaviti svojstva ove klase pre poziva metoda ShowDialog(), što menja ponašanje i izgled ovog dijaloga, ili ograničava datoteke koje se mogu otvoriti. U sledećoj sekciji, pogledaćemo moguće ispravke.

*Kada koristimo klasu **OpenFileDialog** sa konzolnim aplikacijama, sklop **System.Windows.Forms** mora da bude u referencama, a imenovani prostor System.Windows.Forms mora biti uvršćen. Sa Windows Forms aplikacijama pisanim u Visual Studio .NET-u, ovo je već učinjeno iz čarobnjaka za kreiranje aplikacije.*

Naslov dijaloga

Podrazumevani naslov za dijalog **OpenFileDialog** je Open. Možete promeniti naslov dijaloga postavljanjem svojstva Title. Open nije uvek najbolje ime ako, na primer, u aplikaciji želite da analizirate dnevničke datoteke da biste proverili neke informacije i da izvršite proračune na njima, ili dobijete veličine datoteka, i posle celokupne obrade koja je potrebna zatvarate datoteke. U ovom slučaju, datoteke ne ostaju dugo otvorene za korisnika, tako da bi naslov Analiza Datoteka bolje pristajao. Sam Visual Studio .NET ima različite naslove za dijaloge za otvaranje datoteka da bi razlikovao tip podataka koji je otvoren: Open project, Open File, Open Solution i Open File from Web.

Ovaj segment koda pokazuje kako se mogu podešavati različiti naslovi.

```
OpenFileDialog dlg = new OpenFileDialog();
dlg.Title ="Otvori Fajl";
dlg.ShowDialog();
```

Navođenje direktorijuma

Podrazumevano, dijalog otvara direktorijum koji je otvoren od strane korisnika kada su zadnji put pokretali aplikaciju, i prikazuje datoteke u tom direktorijumu. Postavljanjem svojstva **InitialDirectory**, menja se ovo ponašanje. Podrazumevana vrednost svojstva Initial-Directory je prazan string koji predstavlja korisnikov direktorijum **My Documents**, i prikazan je prvi put kada je pokrenuta aplikacija. Drugi put kada se dijalog otvori, prikazani direktorijum biće isti onaj kao i za prethodno otvaranu datoteku. Opšti Windows dijalog koji poziva **OpenFileDialog** koristi registracionu bazu da locira ime prethodno otvarane datoteke.

Možete promeniti ovo ponašanje postavljajući svojstvo **InitialDirectory** na putanju direktorijuma pre poziva metoda **ShowDialog()**. Nikad ne bi trebalo da koristite fiksni string direktorijuma jer isti ne mora da postoji i na korisnikovom sistemu. Da dobijete specijalne sistemske direktorijume možete da koristite statički metod **GetFolderPath()** klase System. Environment. Metod **GetFolderPath()** prihvata enumeraciju **Environment.SpecialFolder**, gde možete definisati za koji sistemski direktorijum želite da vam se vrati putanja.

U sledećem primeru koda, koristimo opšti korisnički direktorijum za šablone da ga postavimo kao **InitialDirectory**:

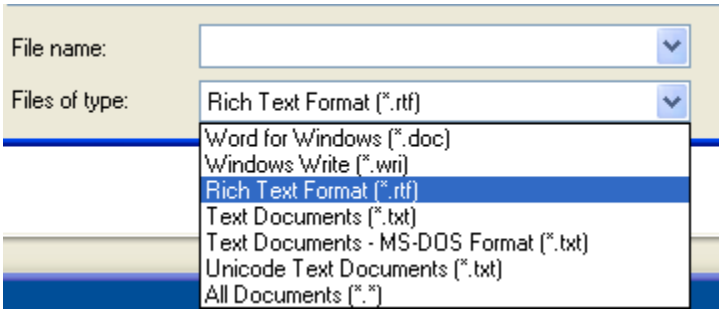
```
string dir =Environment.GetFolderPath(Environment.SpecialFolder.Templates);
dlg.InitialDirectory = dir;
```

Postavljanje filtera za datoteke

Ovaj filter definiše tipove datoteka koje korisnik može izabrati da otvori. Jednostavan filter string može da izgleda ovako:

Text Documents (*.txt | *.txt|All Files | *.*

Ovaj filter se koristi da prikaže ulaze u okviru sa listom **Files of type**. Microsoft WordPad prikazuje ove ulaze:



Filter ima višestruke segmente koji su odvojeni znakom **I** .Dva stringa uvek pripadaju zajedno, tako da broj segmenata uvek treba da bude paran broj. Prvi string definiše tekst koji će biti prikazan u okviru sa listom; drugi string se koristi da naglasi oznake za tip datoteka koje će prikazati u dijalogu. Postavljamo filter string sa svojstvom Filter, kao što se može videti u kodu ispod:

```
dlg.Filter = "Text documents (*.txt) | *.txt | All files (*.*) |*.*";
```

Postavljanje pogrešne vrednosti Filter rezultuje izuzetkom u vreme izvršavanja, **System.ArgumentException**, sa sledećom porukom o grešci: **The provided filter string is invalid.** Prazno mesto pre ili posle filtera takođe nije dozvoljeno.

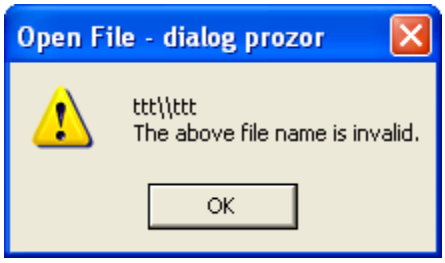
Svojstvo **FilterIndex** definiše podrazumevani izbor u okviru sa listom. Sa WordPadom podrazumevan je **Rich Text Format (*.rtf)** kao što je naglašeno u slici iznad, što može biti teško za čitanje ali vidi se kada se otvori WordPad. Ako imate višestruke tipove datoteka iz kojih treba da birate, onda možete podesiti **FilterIndex** na podrazumevani tip datoteka. Obratite pažnju da FilterIndex ima za osnovu **1**.

Validacija

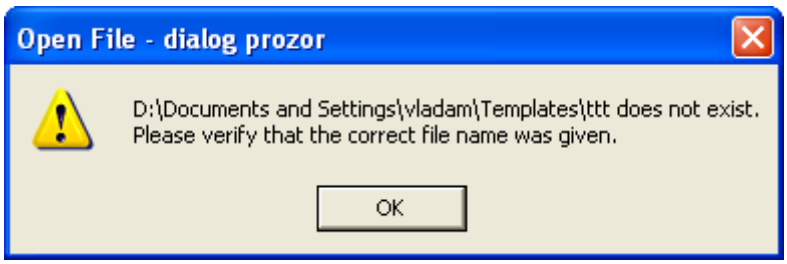
Klasa OpenFileDialog može da radi nešto automatske validacije datoteke pre nego što pokušate da je otvorite. Sa **ValidateNames**, ime datoteke se proverava da se vidi da li je ispravno

Windows ime. Pritiskanje dugmeta OK sa neispravnim imenom izbacuje sledeći dijalog, i korisnik ne može da napusti **OpenFileDialog** pre nego što ispravi ime datoteke. Neispravno ime sadrži neispravne znake kao što su \\, /, ili: **i**.

Unošenje neispravnog imena **ttt\\ttt** rezultuje sledećom porukom o grešci:



CheckFileExists i **CheckPathExists** su dodatna svojstva za validaciju. Sa CheckPathExists proverava se putanja, a **CheckFileExists** - datoteka. Ako datoteka ne postoji, sledeći dijalog se pokazuje kada se pritisne dugme OK:



Podrazumevano za ova tri svojstva je true, tako da se validacija dešava automatski.

Pomoć

Klasa **OpenFileDialog** podržava dugme za pomoć koje je podrazumevano nevidljivo. Podešavanje svojstva **ShowHelp** na true čini ovo dugme vidljivim, i možete dodati rutinu za obradu događaja **HelpRequest** da prikaže pomoć korisniku.

Rezultati

Metoda **ShowDialog()** klase **OpenFileDialog** vraća enumeraciju **DialogResult**. Enumeracija **DialogResult** definiše članove **Abort**, **Cancel**, **Ignore**, **No, None, OK, Retry i Yes. None** je podrazumevana vrednost koja se postavlja sve dok korisnik ne zatvori dijalog. U zavisnosti od pritisnutog dugmeta, odgovarajući rezultat se vraća. Sa klasom **OpenFileDialog**, vraćaju se samo **DialogResult**. **OK** i **DialogResult**. **Cancel**.

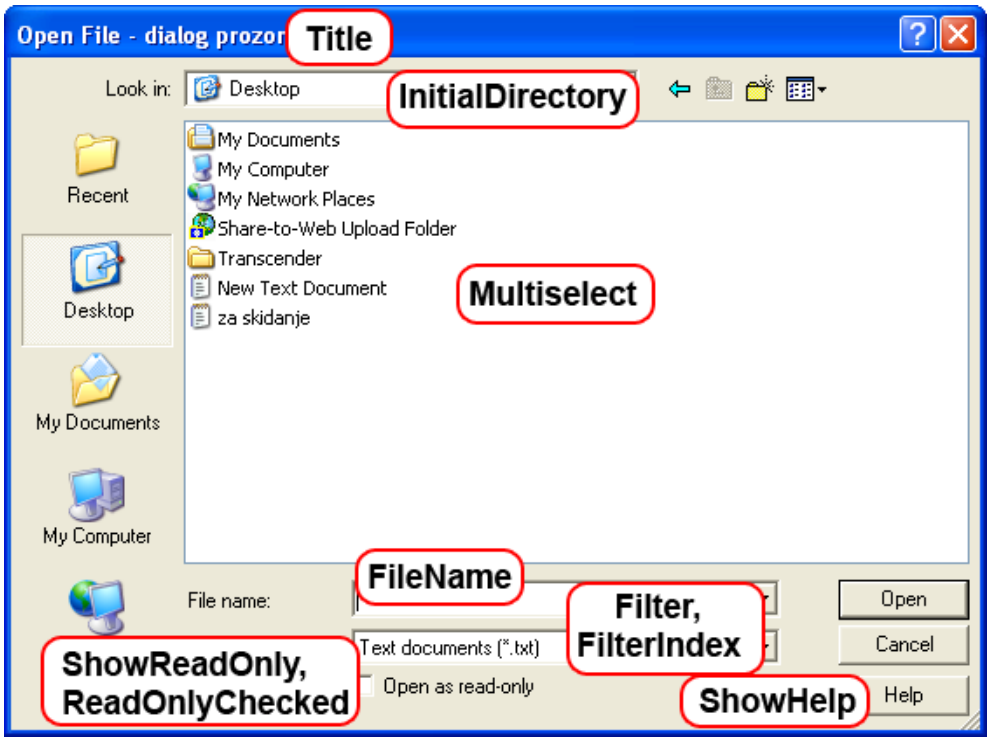
Ako je korisnik pritisnuo dugme OK, izabranom imenu datoteke može da se pristupi koristeći svojstvo **FileName**. Ako je korisnik poništio dijalog, **FileName** je samo prazan string. Ako je svojstvo **MultiSelect** postavljeno na true tako da korisnik može da izabere vise od jedne datoteke, dobijate sva imena datoteka pristupom svojstvu FileNames, koje vraća niz stringova. Ovaj mali deo koda pokazuje kako višestruka imena datoteka mogu da budu dobijena iz dijaloga OpenFileDialog:

```
private void btnOpen_Click(object sender, System.EventArgs e)
{
 OpenFileDialog dlg = new OpenFileDialog();
 dlg.Multiselect=true;
 dlg.Title ="Open File - dialog prozor";
 dlg.ShowReadOnly=true;
 dlg.ShowHelp=true;
 dlg.Filter = "Text documents (*.txt)|*.txt|All files (*.*)|*.*";

 string dir =Environment.GetFolderPath(Environment.SpecialFolder.Templates);

 dlg.InitialDirectory = dir;

 if (dlg.ShowDialog() == DialogResult.OK)
 {
  foreach (string s in dlg.FileNames)
  {
   Console.WriteLine(s);
  }
 }

}
```

Metod **ShowDialog()** otvara dijalog. Svojstvo **Multiselect** je postavljeno na **true**, korisnik može da bira višestruke datoteke. Pritiskom na dugme OK dijaloga isti se završava, ako sve prode dobro i **DialogResult**. OK bude vraćen. Sa **foreach** iskazom, prolazimo kroz sve stringove u nizu stringova koji su vraćeni u svojstvu FileNames i prikazujemo svaku izabranu datoteku.

Svojstva klase OpenFileDialog

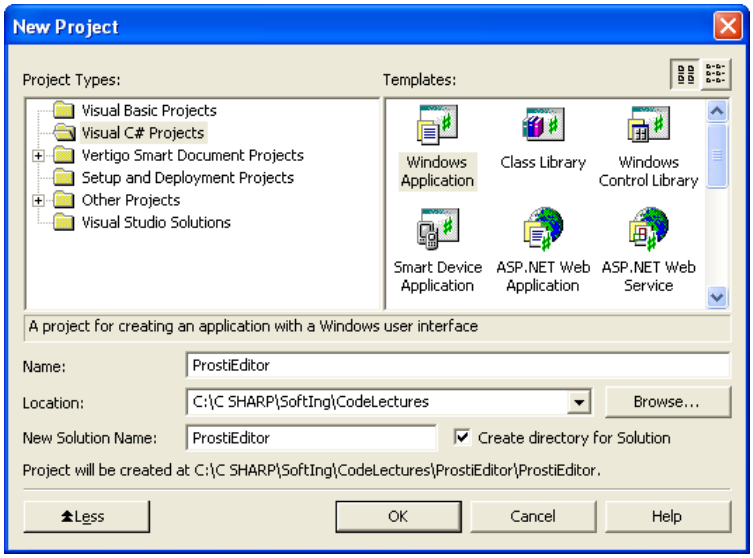
U sledećoj slici možemo da vidimo klasu OpenFileDialog sa svim mogućim svojstvima koja omogućuju uključivanje elemenata prozora. Na slici lako možete da vidite koja svojstva utiču na koje elemente korisničkog interfejsa:



Vežba br. 21.

Kreiranje Windows aplikacije

1. Da bismo se oprobali na otvaranju i zatvaranju datoteka, napravićemo jednostavnu Windows aplikaciju **ProstiEditor**:



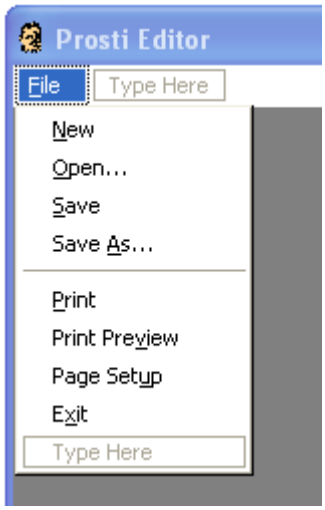
2. Promenite generisanu datoteku **Form1.cs** u **ProstaForma.cs** i klasu **Form1** u **frmMain**. Takođe, menjamo imenovani prostor u **Prosti.Editor**.
Kada se menja ime klase, takođe morate promeniti implementaciju metode **Main()** tako da odražava promenu u imenu, jer se ovo ne radi automatski postavljanjem svojstva **Name** klase **Form**:

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

    Application.Run(new frmMain());

}
```

3. Promenite svojstvo **Text** formulara u Prosti Editor. Okvir za unos teksta sa vise redova će biti površina za čitanje i menjanje podataka iz datoteke, tako da dodajete kontrolu TextBox iz okvira sa alatima u dizajneru Windows formulara. Okvir sa tekstom treba da ima vise redova i trebalo bi da prekriva celu površinu aplikacije, prema tome postavite sledeća svojstva na određene vrednosti.

Osobina (Properties)	Vrednost (Value)
(Name)	txtBoxEdit
Text	<empty>
Multiline	True
Dock	Fill
ScrollBars	Both
AceeptsReturn	True
AcceptsTab	True

4. Sledeće, potrebna nam je kontrola MainMenu za aplikaciju. Glavni meni treba da ima ulaz File sa podmenijima **New**, **Open**, **Save** i **Save As**, kako se vidi na sledećoj slici:



Da bi kod bio čitljiviji, promenite imena u vrednosti u sledećoj tablici. Svojstvo **Text** je prikazano u meniju. **&** znak određuje da će sledeći znak biti podvučen, ako se tastatura koristi za pristup meniju pomoću tastera **<ALT>** i podvučenog znaka. Znak . . . u svojstvu **Text Open** i **Save As** ulaza menija znači da će korisnik biti upitan za podatke pre nego što se akcija dogodi - dijalog će biti otvoren. Kada birate menije **File, New i Save**, akcija se događa bez dodatne intervencije.

*Da se ukratko podsetimo kako možete dodati podulaze menija. Čim upišete tekst **&File** u ulazu glavnog menija, podmeni se otvara ispod njega, u kojem možete da upisujete podulaze u okvir **Type Here**.*

Ime stavke menija	Text
miFile	&File
miFileNew	&New
miFileOpen	&Open...
miFileSave	&Save
miFlleSaveAs	Save &As...

5. Rutina za ulaz glavnog menija &New treba da obrise podatke iz okvira za tekst pozivom metode Clear() kontrole TextBox:

```
private void miFileNew_Click(object sender, System.EventArgs e)
{
  fileName = "Untitled";
  txtBoxEdit.Clear();


}
```

6. Takođe, promenljiva članica **fileName** treba da bude postavljena na **Untitled**. Moramo da deklarišemo i inicijalizujemo ovu promenljivu članicu u klasi **frmMain**:

```
public class frmMain : System.Windows.Forms.Form
{
  private System.Windows.Forms.MainMenu mainMenu1;
  private System.Windows.Forms.MenuItem miFile;
  private System.Windows.Forms.MenuItem miFileNew;
  private System.Windows.Forms.MenuItem miFileOpen;
  private System.Windows.Forms.MenuItem miFileSave;
  private System.Windows.Forms.MenuItem miFileSaveAs;
  /// <summary>
  /// Required designer variable.
  /// </summary>
  private System.ComponentModel.Container components = null;
  private System.Windows.Forms.OpenFileDialog dlgOpenFile;
  private System.Windows.Forms.SaveFileDialog dlgSaveFile;
  private System.Windows.Forms.MenuItem menuItem1;
  private System.Windows.Forms.MenuItem miFilePrint;
  private System.Windows.Forms.MenuItem miFilePrintPreview;
  private System.Windows.Forms.MenuItem miFilePageSetup;
  private System.Windows.Forms.MenuItem miFileExit;
  private System.Drawing.Printing.PrintDocument printDocument;
```

*U meniju **Open** želimo da prikažemo dijalog za otvaranje datoteka, sto ćemo sledeće i uraditi.*

Čitanje datoteke

Sa Prostim Editorom trebalo bi da je moguće predati ime datoteke kao argument dok se pokreće aplikacija. Ime datoteke koje se prosleđuje treba da se koristi za otvaranje i prikazivanje iste u okviru za tekst.

1. Promenite implementaciju metode **Main()** tako da joj se može proslediti argument:

```
[STAThread]
static void Main(string[] args)
{
    string fileName = null;
    if (args.Length != 0)
      fileName = args[0];
    Application.Run(new frmMain(fileName));


}
```

2. Sada moramo da promenimo implementaciju konstruktora **frmMain** da koristi string:

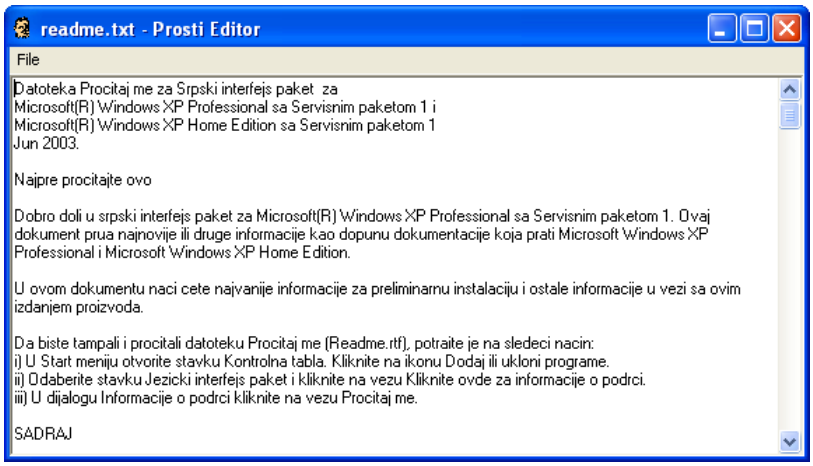
```
public frmMain(string fileName)
{
 //
 // Neophodno za Windows Form Designer podrsku
 //

 //
 // TODO: Add any constructor code after InitializeComponent call
 //
}
```

3. Moramo i da implementiramo metodu **OpenFile()** koja otvara datoteku i ispunjava TextBox podacima iz nje:

```
protected void OpenFile()
{
  try
  {
    using (StreamReader reader = File.OpenText(fileName))
    {
      txtBoxEdit.Clear();
      txtBoxEdit.Text = reader.ReadToEnd();

    }
  }
  catch (IOException ex)
  {
    MessageBox.Show(ex.Message, "Prosti Editor",
      MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
  }
}
```

Ovde koristimo klase StreamReader i File za čitanje datoteke. Da bismo koristili ove klase bez dodavanja imenovanog prostora svaki put, moramo dodati iskaz using System. 10 na početku našeg programa.

4. Sada možemo da pokrenemo aplikaciju iz komandne linije gde predajemo naziv datoteke. Datoteka će biti trenutno otvorena i prikazana u okviru za tekst, kao što može da se vidi na slici ispod:



Kako to radi

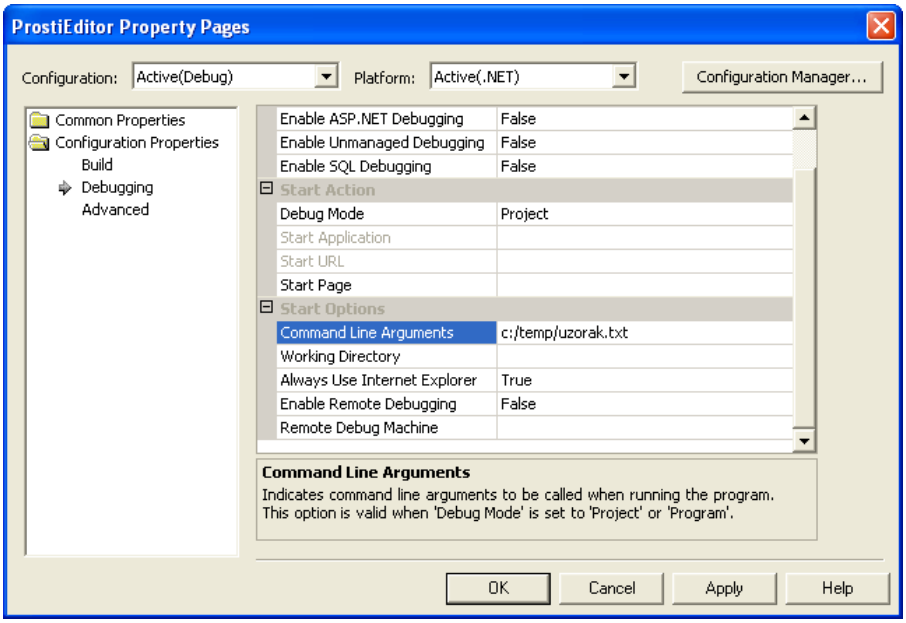
Dodavanjem **string [ ]** u parametre metode **Main()** možemo koristiti bilo koje argumente komandne linije koje je korisnik naveo pri pokretanju aplikacije:

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

U metodi Main() proveravamo da li su argumenti predati, koristeći svojstvo Length. Ako se argument predaje, prvi argument se upisuje u promenljivu fileName koja se koristi za prosleđivanje konstruktoru frmMain:

```
[STAThread]
static void Main(string[] args)
{
    string fileName = null;
    if (args.Length != 0)
      fileName = args[0];
    Application.Run(new frmMain(fileName));


}
```

Unutar Visual Studio .NET-a moguće je deformisati parametre komandne linije za otklanjanje grešaka. U **Solution Exploreru** morate samo izabrati projekat i izabrati **Project\Properties**. Ako izaberete **Configuration Properties \ Debugging** u levom stablu, pojavljuje se dijalog koji je pokazan ispod i možete uneti argumente komandne linije.



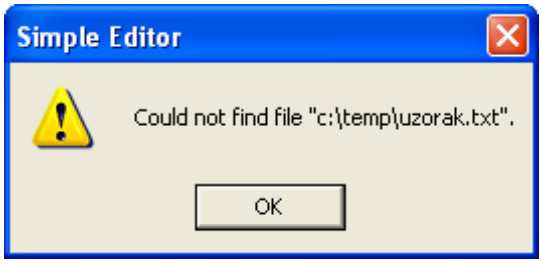
U konstruktoru frmMain proveravamo da li je vrednost promenljive fileName već postavljena. Ako jeste, postavljamo promenljivu članicu fileName i pozivamo metodu OpenFile() da otvori datoteku. Koristimo različit metod **OpenFile()** i ne stavljamo pozive za otvaranje datoteke i popunjavanje kontrole **TextBox** direktno u konstruktoru klase jer **OpenFile()** može da se koristi ponovo u drugim delovima programa.

```
if (fileName != null)
{
  this.fileName = fileName;
  OpenFile();
}
```

U metodi OpenFile() koristimo statički metod OpenText() klase File da bismo otvorili datoteku i dobili vraćen objekat **StreamReader**. Klasa **StreamReader** se onda koristi da se pročita datoteka pomoću metode **ReadToEnd()** koja učitava tekst kao string koji je prosleđen objektu TextBox. Objekat StreamReader bi trebalo da se zatvori posle upotrebe da bi se oslobodili upravljani i neupravljani resursi. Ovo ćemo učiniti iskazom using. Tako using poziva **Dispose()** na kraju bloka, a **Dispose()** implementacija klase StreamReader na kraju poziva **Close()** da zatvori datoteku:

```
{
  using (StreamReader reader = File.OpenText(fileName))
  {
    txtBoxEdit.Clear();
    txtBoxEdit.Text = reader.ReadToEnd();
```

Pošto operacije sa datotekama mogu lako da generišu izuzetke, na primer kada korisnik nema odgovarajuća prava pristupa datoteci, ili kad sama datoteka ne postoji, kod je stavljen unutar bloka **try**. U slučaju IO izuzetka, pokazuje se okvir sa porukom korisniku da ga informiše o problemu, ali aplikacija i dalje radi:
Ako upišemo nepostojeće ime datoteke kada pokrenemo aplikaciju, pojavljuje se ovaj okvir sa porukom:



Dodavanje dijaloga za otvaranje datoteke
Sad možemo da čitamo datoteke sa ovim jednostavnim programom za uređivanje, predajući ime datoteke pri pokretanju aplikacije. Ali naravno, koristimo opšte dijalog klase koje ćemo dodati odmah aplikaciji.

1. U kategoriji Windows **Forms** okvira sa alatima možemo da nađemo komponentu OpenFileDialog. Dodajte ovu komponentu u dizajner Windows formulara. Menjamo samo tri svojstva: ime instance u **dlgOpen**, svojstvo Filter će biti postavljeno na sledeći string, a svojstvo **FilterIndex** je postavljeno na 2 da bi Pupin dokumenti bili podrazumevani izbor:

Kako to radi

Dodavanjem komponente OpenFileDialog u dizajner Windows formulara, novi privatni član je dodat klasi frmMain:

```
public class frmMain : System.Windows.Forms.Form
{
    private System.Windows.Forms.MainMenu mainMenu1;
    private System.Windows.Forms.MenuItem miFile;
    private System.Windows.Forms.MenuItem miFileNew;
    private System.Windows.Forms.MenuItem miFileOpen;
    private System.Windows.Forms.MenuItem miFileSave;
    private System.Windows.Forms.MenuItem miFileSaveAs;
    /// <summary>
    /// Required designer variable.
    /// </summary>
    private System.ComponentModel.Container components = null;

    private System.Windows.Forms.SaveFileDialog dlgSaveFile;
    private System.Windows.Forms.MenuItem menuItem1;
```

U dizajneru koda Windows formulara, **initializeComponent()**, nova instanca klase OpenFileDialog se kreira i odgovarajuća svojstva se postavljaju. Da bi se video sledeći kod u programu za uređivanje koda, morate da pritisnete + znak linije **Windows Forms Designer generated code** i onda + znak linije private void **InitializeComponent()** :

```
private void InitializeComponent()
{
 System.Resources.ResourceManager resources = new System.Resources.Re
 this.mainMenu1 = new System.Windows.Forms.MainMenu();
 this.miFile = new System.Windows.Forms.MenuItem();
 this.miFileNew = new System.Windows.Forms.MenuItem();
 this.miFileOpen = new System.Windows.Forms.MenuItem();
 this.miFileSave = new System.Windows.Forms.MenuItem();
 this.miFileSaveAs = new System.Windows.Forms.MenuItem();
 this.menuItem1 = new System.Windows.Forms.MenuItem();
 this.miFilePrint = new System.Windows.Forms.MenuItem();
 this.miFilePrintPreview = new System.Windows.Forms.MenuItem();
 this.miFilePageSetup = new System.Windows.Forms.MenuItem();
 this.miFileExit = new System.Windows.Forms.MenuItem();
 this.menuItem2 = new System.Windows.Forms.MenuItem();
 this.miFont = new System.Windows.Forms.MenuItem();
 this.miColor = new System.Windows.Forms.MenuItem();

 this.dlgSaveFile = new System.Windows.Forms.SaveFileDialog();
 this.printDocument = new System.Drawing.Printing.PrintDocument();
....
....
 //
 // dlgOpenFile
 //
```

Sa podrškom dizajnera Windows formulara, pravimo novu instancu klase OpenFileDialog i postavljamo njena svojstva. Sada smo spremni da prikažemo dijalog.

Prikazivanje dijaloga OpenFileDialog
Dodajte rutinu za obradu događaja Click ulaza u meni Open, u kojoj prikazujemo dijalog i čitamo izabranu datoteku sa sledećim kodom:

```
private void miFileOpen_Click(object sender, System.EventArgs e)
{
  if (dlgOpenFile.ShowDialog() == DialogResult.OK)
  {
    fileName = dlgOpenFile.FileName;
    OpenFile();

  }

}
```

Kako to radi

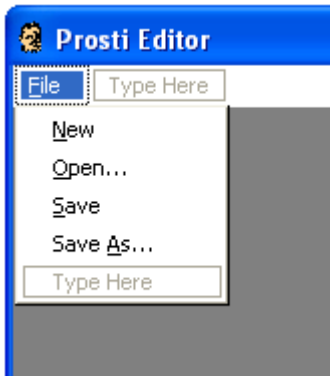
Metoda **ShowDialog()** prikazuje dijalog za otvaranje datoteka i vraća dugme koje je korisnik pritisnuo. Ništa nećemo uraditi sve dok korisnik ne pritisne dugme OK. To je razlog zašto proveravamo DialogResult .OK u iskazu if. Ako korisnik poništi dijalog, ništa nećemo uraditi:

```
  if (dlgOpenFile.ShowDialog() == DialogResult.OK)
  {
```

Zatim ćemo uzeti ime izabrane datoteke pristupajući svojstvu FileName klase OpenFileDialog i postavljajući promenljivu članicu fileName na ovu vrednost. Ovo je vrednost koja se koristi u metodi OpenFile. Takođe bi bilo moguće direktno otvoriti datoteku sa klasom OpenFileDialog pozivajući metod **dlgOpenFile.OpenFile()** koji nam već vraća objekat Stream, ali pošto već imamo metod OpenFile koji otvara i čita datoteku, koristićemo njega:

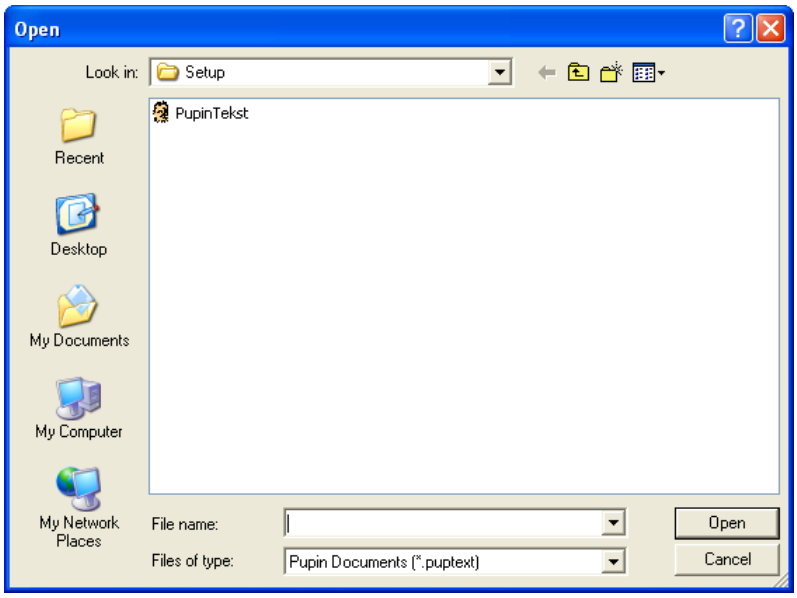
```
{
  fileName = dlgOpenFile.FileName;
  OpenFile();

}
```

Pokretanje Prostog Editora

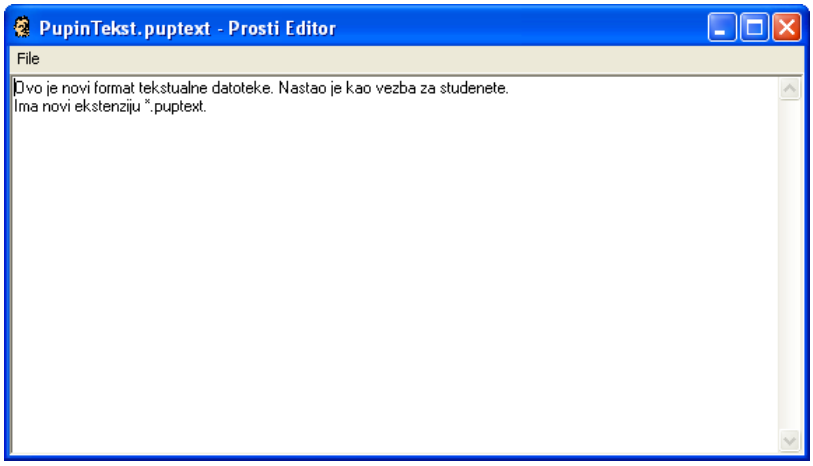


1. Sada možemo da pokrenemo naš jednostavni program za uređivanje teksta. Samo ulazi menija New i Open trenutno rade. Save i Save As... će biti implementirani u sledećoj sekciji.

Izborom ulaza u meni **File | Open...** pojavljuje se OpenFileDialog i možemo izabrati datoteku. Pretpostavlja se da trenutno nemate datoteke sa oznakom za tip **.puptext**. U ovom trenutku ne možemo da snimamo datoteke, pa zato izaberite drugačiji tip datoteke u dijalogu za otvaranje datoteke, ili možete iskopirati tekst datoteku tako da ima oznaku za tip **.puptext**:



Izaberite tekst datoteku - pritiskanjem dugmeta Open tekst se pojavljuje u okviru za tekst dijaloga.



U ovom momentu već možemo čitati postojeće datoteke. Sada bi bilo odlično ako bismo prešli na stvaranje novih i menjanje starih. Koristićemo SaveFileDialog da bismo ovo učinili.

Klasa SaveFileDialog

Klasa SaveFileDialog je vrlo slična klasi OpenFileDialog i one imaju skup zajedničkih svojstava. Ovde nećemo pričati o onim svojstvima koja operišu na isti način kao kod dijaloga za otvaranje datoteka. Umesto toga, usmerićemo pažnju na svojstva specifična za dijalog za snimanje datoteka, kao i na mesta gde se primena opštih svojstava razlikuje.

Naslov

Pomoću svojstva Title možete postaviti naslov dijaloga slično kao u klasi OpenFileDialog. Ako ništa nije postavljeno, podrazumevani naslov je Save As.

Oznake za tip datoteke

Oznake za tip datoteka se koriste da povežu datoteke sa aplikacijama. Najbolje je dodati oznaku za tip datoteci; u suprotnom, Windows neće biti u mogućnosti da zna koja aplikacija bi trebalo da se iskoristi za otvaranje datoteke, i verovatno je da ćete i vi to zaboraviti.
AddExtension je svojstvo tipa **Boolean** koje definiše da li će oznaka za tip datoteke biti automatski dodata imenu koje korisnik unese. Ako korisnik sam unese oznaku, dodatna oznaka neće biti dodavana. Tako da ako korisnik unese za ime datoteke test, datoteka test. txt će biti upisana na disk. Ako se upiše naziv test.txt, datoteka koja će biti snimljena će i dalje biti test.txt, a ne test.txt.txt.
Svojstvo **DefaultExt** postavlja nastavak za tip datoteke koji će se koristiti ako korisnik ne upiše istu. Ako ostavite svojstvo praznim, oznaka tipa datoteke koja je definisana sa trenutno izabranim svojstvom Filter će biti korišćena umesto njega. Ako postavite i svojstvo Filter i svojstvo **DefaultExt**, DefaultExt će biti korišćen bez obzira na Filter.

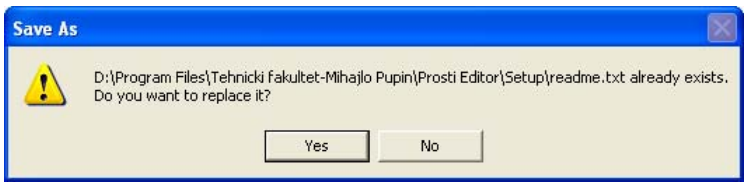
Validacija

Za automatsku validaciju imena datoteke, imamo svojstva **ValidateNames**, **CheckFileExists** i **CheckPathExists**, kao i kod klase **OpenFileDialog**. Razlika između klasa **OpenFileDialog** i **SaveFileDialog** jeste u tome što je za klasu **SaveFileDialog**; podrazumevana vrednost za svojstvo **CheckFIleExists** - **false**, što znači da možete dostavite ime novoj datoteci za snimanje.

Snimanje preko postojećih datoteka

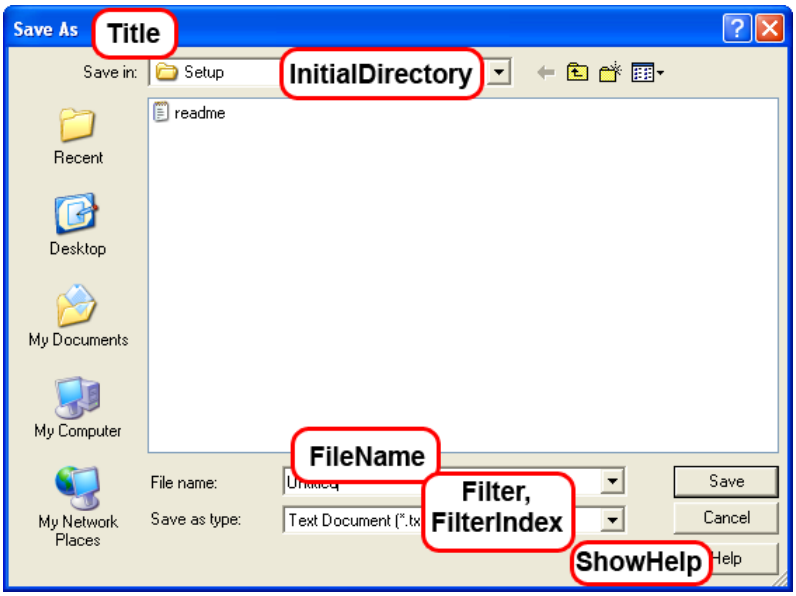
Kao što smo videli, validacija imena datoteka je slična kao kod klase **OpenFileDialog**. Ali ipak za klasu SaveFileDialog postoji još provera koje treba sprovesti i još svojstava za podešavanje. Na prvom mestu, ako postavite svojstvo **CreatePrompt** na **true**, korisnik će biti upitan da li želi da se napravi nova datoteka. Ako se svojstvo **OverwritePrompt** postavi na **true**, to znači da je korisnik upitan da li zaista želi da presnimi preko postojeće datoteke. Predefinisana vrednost za

svojstvo **OverwritePrompt** je **true**, a za **CreatePrompt** je **false**. Sa ovim podešavanjem sledeći dijalog se prikazuje ako korisnik želi da snimi preko postojeće datoteke:



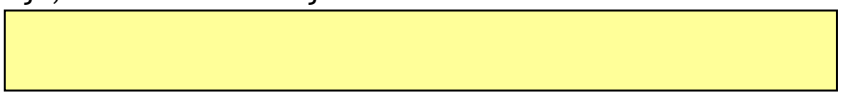
Svojstva klase FileSaveDialog

Kao i u prethodnom dijalogu, ovde ponovo imamo sliku koja sumira sva svojstva klase FileSaveDialog:



Dodavanje dijaloga SaveFileDialog

1. Možemo da dodamo dijalog **SaveFileDialog** sličan dijalogu **OpenFileDialog** u dizajneru Windows formulara: izaberite komponentu SaveFileDialog iz okvira sa alatima i spustite je na formular. Promenite svojstvo **Name** u **dlgSaveFile**, FileName u **Untitled**, **FilterIndex** u **2**, i **Filter** u sledeći string kao što smo radili ranije sa **OpenFileDialog**. Pošto samo želimo da dopustimo oznake tipova datoteka **.txt** i **.puptext** da budu snimljene sa ovim programom za uređivanje, ***.*** će biti izostavljeno:



2. Dodajte rutinu za obradu događaja **Click** za ulaz menija **Save As** i dodajte sledeći kod. U ovom kodu prikazaćemo dijalog **SaveFileDialog** pomoću metode **ShowDialog()**. Kao i sa dijalogom **OpenFileDialog**, samo nas interesuje rezultat ako je korisnik pritisnuo dugme OK. Pozvaćemo metod SaveFile() koji snima datoteku na disk. Ova metoda mora da bude implementirana u sledećem koraku:

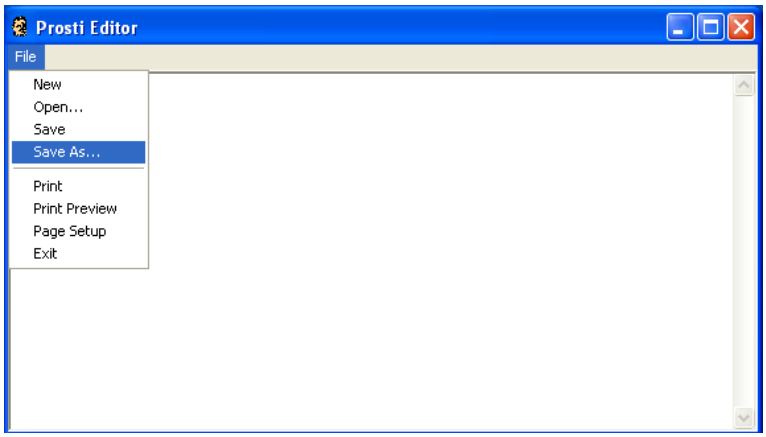
```
private void miFileSaveAs_Click(object sender, System.EventArgs e)
{
  if (dlgSaveFile.ShowDialog() == DialogResult.OK)
  {
    fileName = dlgSaveFile.FileName;
    SaveFile();

  }

}
```

3. Dodajte metod SaveFile(), kao što se može videti ovde, u vašu datoteku:

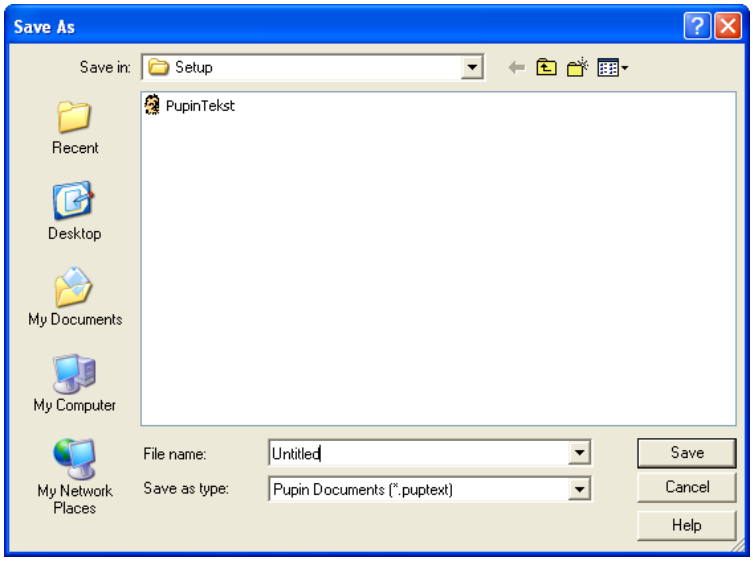
```
protected void SaveFile()
{
  try
  {
    Stream stream = File.OpenWrite(fileName);
    using (StreamWriter writer = new StreamWriter(stream))
    {
      writer.Write(txtBoxEdit.Text);
    }
  }
  catch (IOException ex)
  {
    MessageBox.Show(ex.Message, "Prosti Editor",
      MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
  }
}
```

Slično metodi OpenFile, koristićemo klasu File da otvorimo datoteku, ali sada je otvaramo za upis sa **OpenWrite()**. Metod **OpenWrite()** vraća objekat Stream koji je prosleđen konstruktoru klase StreamWriter. Metoda **Write()** klase **StreamWriter** upisuje sve podatke iz kontrole TextBox u datoteku. Na kraju bloka **using**, objekat **StreamWriter** se zatvara. Stream objekat je povezan sa objektom writer; stream se takođe zatvara, i nema potrebe za pozivom dodatnog metoda Close() objekta stream.

4. Posle izgradnje projekta, možemo pokrenuti aplikaciju koristeći meni **Debug | Start Visual Studio .NET-a**. Napišite nešto teksta u TextBox i izaberite meni File | Save As... kao što je pokazano na slici pored:



Dijalog SaveFileDialog će se pojaviti, kao što je pokazano ispod. Sada možete snimiti datoteku i ponovo je otvoriti da napravite još izmena:



Implementiranje rutine za snimanje datoteke
Možemo da izvedemo opciju Save As, ali jednostavni Save nije dostupan u ovom trenutku. Dodajte rutinu za obradu događaja Click stavke menija Save i dodajte sledeći kod:

```
private void miFileSave_Click(object sender, System.EventArgs e)
{
  if (fileName == "Untitled")
  {
    miFileSaveAs_Click(sender, e);
  }
  else
  {
    SaveFile();
  }
}
```

Kako to radi

Sa menijem Save, datoteka bi trebalo da bude snimljena bez otvaranja dijaloga. Postoji jedan izuzetak ovom pravilu, a to je da ako se kreira novi dokument, a korisnik nije naveo ime datoteke, tada rutina za snimanje treba da radi kao što to čini Save As i prikaže dijalog za snimanje datoteke.

Sa promenljivom članicom **fileName** lako možemo da proverimo da li je datoteka otvorena ili je ime još postavljeno na inicijalnu vrednost Untitled posle kreiranja novog dokumenta. Ako iskaz if vraća true, pozivamo rutinu **miFileSaveAs_Click()** koju smo prethodno implementirali za meni Save As.

*U drugom slučaju, kada je datoteka otvorena i korisnik sad bira meni Save, izvršavanje prolazi u blok **else**. Možemo koristiti isti metod SaveFile()koji smo prethodno implementirali.*

Postavljanje naslova formulara
1. Sa Notepadom, Wordom i drugim Windows aplikacijama ime dokumenta koji se trenutno uređuje prikazano je u naslovu aplikacije. Trebalo bi i ovu mogućnost da dodamo.
Napravite novu funkciju članicu **SetFormTitle()** i dodajte ovu implementaciju:

```
protected void SetFormTitle()
{
  FileInfo fileinfo = new FileInfo (fileName);
  this.Text = fileinfo.Name + " -  Prosti Editor";
}
```

Klasa **Fileinfo** se koristi da se dobije ime datoteke bez putanje koja prethodi, a nalazi se u promenljivoj filename. Klase **Fileinfo** i **StreamWriter** su pokrivene u nekom drugom predavanju. Dodajte poziv ovog metoda u rutine **miFileNew_Click()**, **miFileOpen_Click()**, i **miFileSaveAs_Click()** posle postavljanja promenljive članice **fileName**, kao što se može videti u sledećim segmentima koda:

```
private void miFileNew_Click(object sender, System.EventArgs e)
{
 fileName = "Untitled";
 txtBoxEdit.Clear();
}

 private void miFileOpen_Click(object sender, System.EventArgs e)
 {
   if (dlgOpenFile.ShowDialog() == DialogResult.OK)
   {
     fileName = dlgOpenFile.FileName;
     OpenFile();
   }
 }

 private void miFileSaveAs_Click(object sender, System.EventArgs e)
 {
   if (dlgSaveFile.ShowDialog() == DialogResult.OK)
   {
     fileName = dlgSaveFile.FileName;
     SaveFile();
   }
 }
```

Kako to radi

Svaki put kada se ime datoteke promeni, svojstvo **Text** aktuelnog formulara će biti promenjeno na ime datoteke na koje je nadovezano ime aplikacije.

Pokrenuvši aplikaciju sada vidite sledeći ekran. Ovde, pošto uredujemo datoteku **uzorak.txt**, ova informacija je prikazana u naslovu formulara:

Sada je naš jednostavni program za uređivanje kompletan: možemo otvarati, praviti i snimati datoteke. Da li smo završili? Ne baš - trebalo bi da dodamo i mogućnost štampanja.

Štampanje

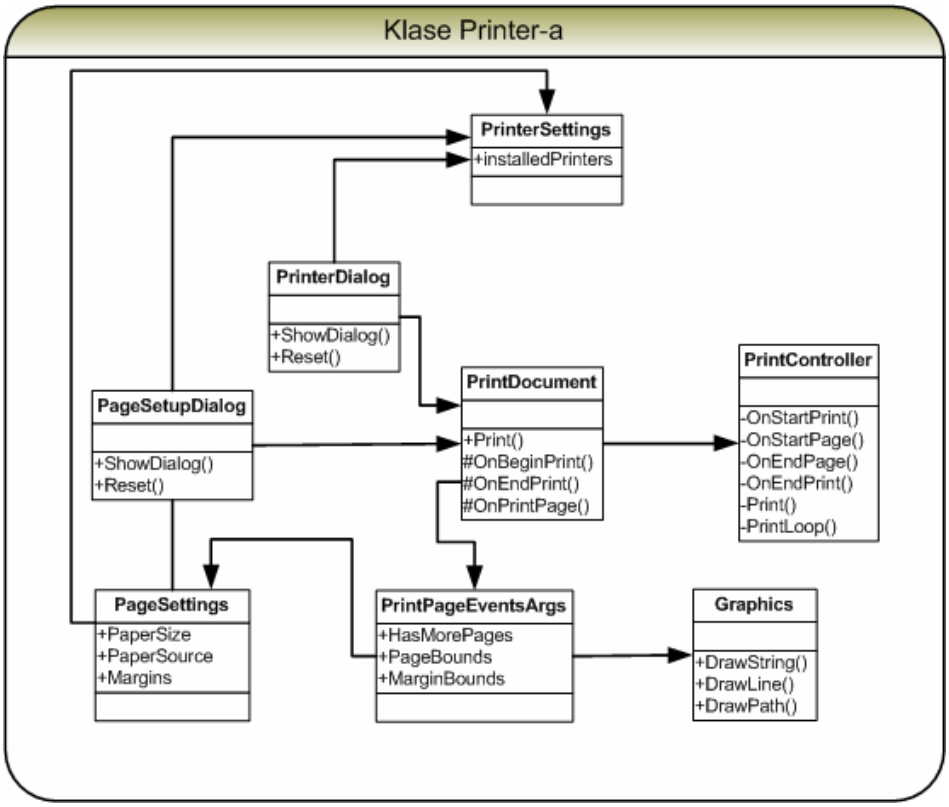
Sa štampanjem, moramo da brinemo o izboru štampača, podešavanjima za stranu, i o tome kako da štampamo višestruke stranice. Sa .NET klasama u imenovanom prostoru **System.Drawing.Printing** zadatak nam je znatno olakšan.

Pre nego što pogledamo klasu PrintDialog koja omogućava izbor štampača, pogledaćemo kako .NET rukuje štampanjem. Osnova za štampanje je klasa **PrintDocument** čiji metod **Print()** započinje lanac poziva koji se završava pozivom **OnPrintPage()**, koji je odgovoran za

predavanje izlaza na štampač. U svakom slučaju, pre nego što uđemo dublje u implementaciju koda za štampu, pogledajmo nešto detaljnije .NET klase za štampanje.

Arhitektura štampanja

Sledeći dijagram pokazuje glavne delove arhitekture štampanja u formularu dijagrama koji razjašnjava odnose između klasa i nekih svojstava i metoda:

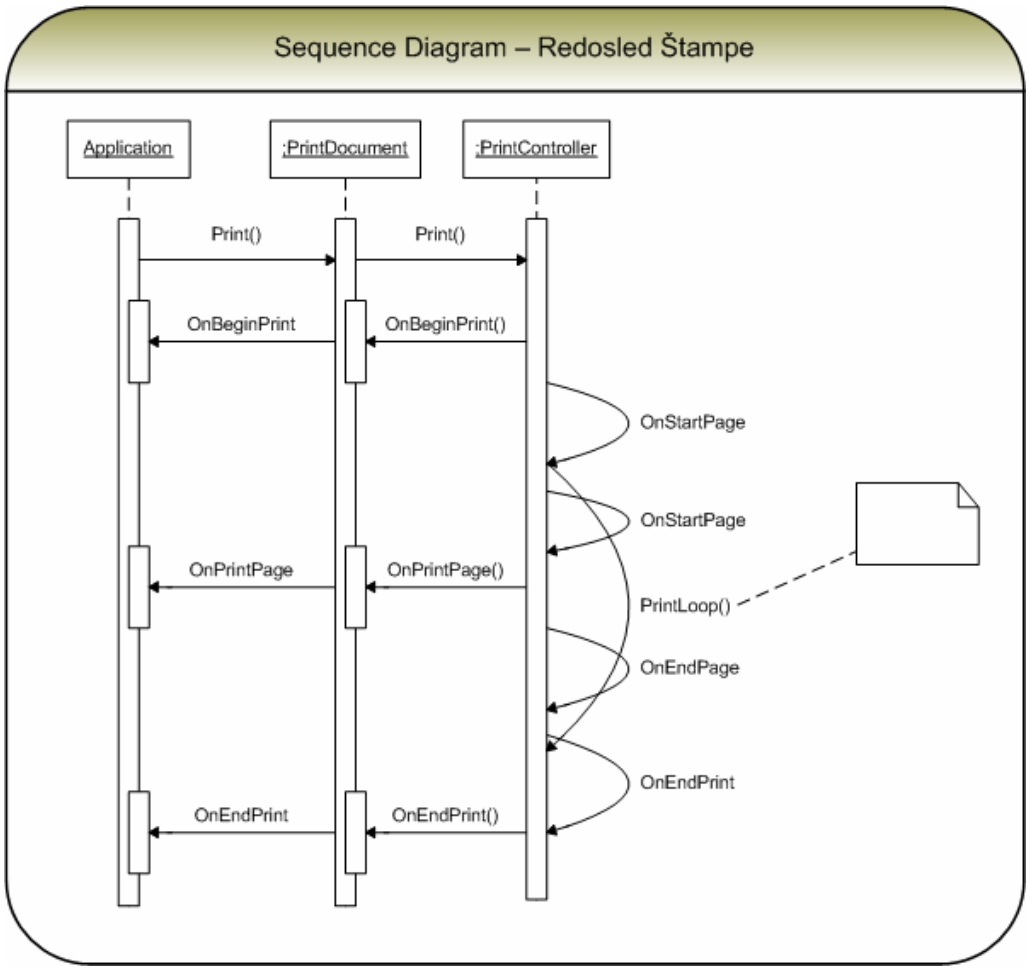


Pogledajmo na funkcionalnost ovih klasa:

- Počnimo sa najvažnijom klasom, PrintDocument. U dijagramu vidite da skoro sve druge klase imaju komunikaciju sa ovom klasom. Da bi se odštampao dokument, instanca klase PrintDocument je neophodna. U drugom dijagramu, pogledaćemo sekvencu štampanja koju inicira ova klasa. Ali pre toga, pogledajmo takođe i druge klase.
- Klasa **PrintController** kontroliše tok zadatka za štampu. Počevši zadatak štampe, kontroler štampe ima događaje za početak štampe, za svaku stranicu i za kraj štampe. Klasa je apstraktna jer se implementacija razlikuje od normalne štampe i pregleda štampanja.
- Možemo dobiti i podesiti konfiguraciju štampača kao što je dvostrana štampa, orijentacija papira (vertikalna ili horizontalna), kao i broj kopija, sa klasom **PrinterSettings**.
- Na kojem štampaču ćemo štampati i kako će **PrinterSettings** biti konfigurisan, posao je klase **PrintDialog**. Ova klasa je izvedena iz klase CommonDialog kao i ostale klase za dijalog sa kojima smo se sretali, i možda već imate ideju kako će biti upotrebljena.
- Klasa **PageSettings** postavlja veličinu i granice strane, i da li je strana u koloru ili crno-bela. Konfiguracija ove klase može da bude izvedena sa klasom PageSetupDialog koja je, takođe, izvedena iz klase CommonDialog.

Redosled štampanja

Sad kada znamo uloge klasa u arhitekturi štampanja, pogledajmo glavni redosled štampe. Dijagram ispod pokazuje glavne učesnike, aplikaciju koju želimo da napišemo, instancu klase **PrintDocument** i **PrintController** u vremenskoj sekvenci:



Aplikacija mora da pozove metod **Print()** klase **PrintDocument**. Ovo počinje sekvencu štampanja. Kako objekat PrintDocument sam nije odgovoran za tok štampe, zadatak je dat objektu **PrintController** pozivanjem metoda **Print()** ove klase.
Kontroler štampe sad preuzima akciju i obaveštava PrintDocument da je štampanje počelo pozivajući metod **OnBeginPrint().** Ako naša aplikacija treba nešto da uradi na početku zadatka za štampu, moramo da registrujemo rutinu za obradu događaja u klasi PrintDocument tako da budemo obavešteni u našoj klasi aplikacije. Na pokazanom dijagramu pretpostavlja se da smo registrovali rutinu **OnBeginPrint()**, tako da se ona poziva iz klase PrintDocument.

Kada je početna faza završena, **PrintControler** ulazi u metod **PrintLoop()** da bi pozvao metod **OnPrintPage()** u klasi PrintDocument za svaku stranu koja treba da se odštampa. Metod **OnprintPage()** poziva sve ratine za obradu događaja **PrintPage**. Moramo da implementiramo takvu ratinu u svakom slučaju; u suprotnom, ništa ne bi bilo odštampano. U dijagramu iznad možete da vidite da se rutina naziva **OnPrintPage()**.
Posle zadnje odštampane stranice, PrintController poziva metod **OnEndPrint()** u klasi **PrintDocument**. Takođe, opciono, možemo da implementiramo rutinu koja treba da bude pozvana na ovom mestu.

Najvažnija stvar koju treba da znamo jeste:

*Možemo da implementiramo kod za štampu u rutini za obradu događaja PrintDocument. PrintPage. Ona će biti pozivana za svaku stranicu koja je će biti odštampana. Ako postoji kod za štampu koji bi trebalo da se poziva samo jednom po zadatku štampe, moramo da implementiramo rutine za obradu događaja **BeginPrint** i **EndPrint**.*

Događaj PrintPage

Sada moramo da implementiramo rutinu za obradu događaja PrintPage. Delegat **PrintPageEventHandler** definiše argumente ove rutine:

Kao što možete videti, dobijamo objekat tipa **PrintPageEventArgs**. Možete da pogledate nazad u dijagramu klasa da proverite glavna svojstva ove klase. Ova klasa ima veze sa klasama **PageSettings** i **Graphics**; prva nam omogućava da postavimo veličinu papira, margine, i da dobijemo informacije o samom štampaču. Klasa **Graphics**, sa druge strane, omogućava nam pristup kontekstu štampača i slanje stringova, linija i krivih.

GDI označava Graphics Device Interface i omogućava nam određeni grafički izlaz na uređaj kao što je ekran i štampač. GDI+je sledeća generacija GDI-ja koja dodaje karakteristike kao što su četke sa više nivoa i alfa pretapanje.

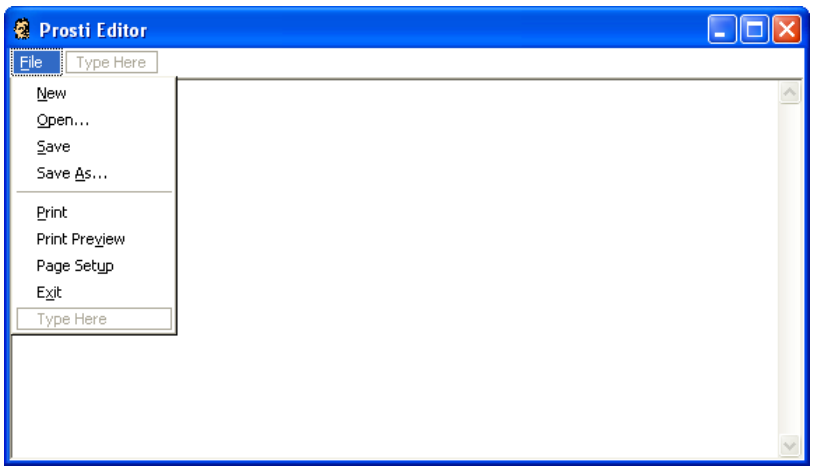
Kroz sledeći primer videćete da je dodavanje mogućnosti štampe aplikacijama prilično lak zadatak.

Dodavanje stavke Print u meni

Pre nego što možemo da dodamo dijalog PrintDialog, moramo da dodamo par ulaza u meni za štampu. Dodajte dva graničnika i stavke menija **Print**, **PrintPreview**, **PageSetup** i **Exit**.
Ovde su navedena svojstva **Name** i **Text** novih stavki menija:

Ime stavke menija	Text
miFilePrint	&Print
miFilePrintPreview	Print Pre&view...
miFilePageSetup	Page Set&up...
miFileExit	E&xit

Meni bi trebalo da izgleda kao na sledećoj slici:



Dodavanje komponente PrintDocument i rutine za obradu događaja

1. Prevucite komponentu **PrintDocument** iz okvira sa alatima i spustite je na formular. Promenite ime u **PrintDocument**, i dodajte rutinu **OnPrintPage()** za obradu događaja **Printpage** izborom dugmeta **Events** u prozoru **Properties**. Tada dodajte ovaj kod implementaciji rutine za obradu događaja:

```
private void OnPrintPage(object sender, System.Drawing.Printing.PrintPageEventArgs e)
{
 char[] param = {'\n'} ;
 string[] lines = txtBoxEdit.Text.Split(param);

 int i=0;
 char[] trimParam = {'\r'} ;
 foreach (string s in lines)
 {
  lines[i++] = s.TrimEnd(trimParam);
 }

 int x = 20;
 int y = 20;

 foreach (string line in lines)
 {
   e.Graphics.DrawString (line, new Font("Arial", 10), Brushes.Black, x, y);
   y += 15;
 }
}
```

2. Zatim dodajte rutinu za obradu događaja **Click** menija **Print** da biste pozvali metod Print() klase PrintDocument:

```
private void miFilePrint_Click(object sender, System.EventArgs e)
{
  printDocument.Print();
}
```

3. Sada možete da izgradite i pokrenete aplikaciju i odštampate dokument. Naravno, morate da imate štampač instaliran da bi primer radio.

Kako to radi

Metod Print() objekta **printDocument** poziva događaj PrintPage objekta **printDocument** uz pomoć klase PrintController klase:

```
printDocument.Print();
```

U rutini **OnPrintPage()**, podelimo tekst unutar okvira za tekst red po red, koristeći metod **String.Split()** i znak za novu liniju **\n**. Rezultujući stringovi su upisani u promenljivu lines:

```
private void OnPrintPage(object sender, System.Drawing.Printing.PrintPageEventArgs e)
{
 char[] param = {'\n'} ;
 string[] lines = txtBoxEdit.Text.Split(param);
```

Zavisno od toga kako je tekst kreiran, redovi nisu samo razdvojeni znakom **\n** , već i znakom **\r**. Sa metodom **TrimEnd()** string klase, znak **\r** je otklonjen iz svakog stringa:

```
int i=0;
char[] trimParam = {'\r'} ;
foreach (string s in lines)
{
 lines[i++] = s.TrimEnd(trimParam);
}
```

U drugom iskazu **foreach** u kodu ispod, možete videti da prolazimo kroz sve linije i šaljemo svaku liniju na štampač pozivom metoda **e.Graphics.Drawstring()**. **e** je promenljiva tipa **PrintPageEventArgs** gde je svojstvo Graphics povezano sa kontekstom štampača. Kontekst štampača omogućuje crtanje na uređaju za štampanje; klasa Graphics ima svoje metode za iscrtavanje u ovom kontekstu.

Pošto još ne možemo da izaberemo štampač, koristimo podrazumevani štampač, čiji detalji se čuvaju u Windows registracionoj bazi.
Sa metodom **Drawstring()** koristimo font **Arial** sa veličinom od **10 tačaka** i crnom linijom na izlazu štampača. Pozicija za izlaz je definisana promenljivama **x** i **y**. Horizontalna pozicija je fiksirana na **20 piksela**; vertikalna pozicija se povećava za svaki red:

```
        int x = 20;
        int y = 20;

        foreach (string line in lines)
        {
          e.Graphics.DrawString (line, new Font("Arial", 10), Brushes.Black, x, y);
          y += 15;
        }
      }
```

Štampa koju smo dosad uradili nije bez greške:

- Izlaz štampe je poslat na podrazumevani štampač kao što je postavio korisnik u Control Panelu. Bilo bi bolje za našu aplikaciju da korisnik može da bira štampač. Koristićemo klasu **PrintDialog** za ovo.
- Font je fiksiran. Da bi se omogućilo korisniku da izabere font, možemo koristiti klasu **FontDialog** koju ćemo posle pogledati detaljnije.
- Okviri strane su fiksirani na unete vrednosti u našem programu. Da bi korisnik postavio vrednosti za različite okvire strane, koristićemo klasu **PageSetupDialog**.
- Štampanje vise strana ne radi. Ako se dokument za štampu prostire na vise strana, samo prva strana se štampa. Takođe bi bilo lepo da se i zaglavlje (na primer, ime datoteke) i fusnota (na primer, broj strane) štampaju.

Prema tome, nastavimo sa procesom štampe da bismo sredili ove stavke.

Štampanje više strana

Događaj **Printpage** biva pozivan za svaku stranu štampe. Moramo samo da obavestimo objekat PrintController da trenutna strana u štampi nije zadnja strana, podešavajući svojstvo **HasMorePages** klase **PrintPageEventArgs** na vrednost **true**.

Izmena metoda OnPrintPage za štampanje vise strana
1. Morate takođe deklarisati promenljivu članicu lines tipa **string[ ]** i promenljivu **linesPrinted** tipa **int** u klasi **frmMain**:

```
public class frmMain : System.Windows.Forms.Form
{
   private System.Windows.Forms.MainMenu mainMenu1;
   private System.Windows.Forms.MenuItem miFile;
   private System.Windows.Forms.MenuItem miFileNew;
   private System.Windows.Forms.MenuItem miFileOpen;
   private System.Windows.Forms.MenuItem miFileSave;
   private System.Windows.Forms.MenuItem miFileSaveAs;
  /// <summary>
  /// Required designer variable.
  /// </summary>
   private System.ComponentModel.Container components = null;
   private System.Windows.Forms.OpenFileDialog dlgOpenFile;
   private System.Windows.Forms.SaveFileDialog dlgSaveFile;
   private System.Windows.Forms.MenuItem menuItem1;
   private System.Windows.Forms.MenuItem miFilePrint;
   private System.Windows.Forms.MenuItem miFilePrintPreview;
   private System.Windows.Forms.MenuItem miFilePageSetup;
   private System.Windows.Forms.MenuItem miFileExit;
   private System.Drawing.Printing.PrintDocument printDocument;
   private string fileName = "Untitled";
```

2. Promenite rutinu **OnPrintPage()**. U prethodnoj implementaciji **OnPrintPage()** podelili smo tekst u linije. Pošto se metod **OnPrintPage()** poziva za svaku stranicu, deljenje teksta u redove je potrebno samo jednom, na početku naše operacije štampe. Uklonite sav kod iz ratine **OnPrintPage()** i zamenite ga novom implementacijom:

```
private void OnPrintPage(object sender, System.Drawing.Printing.PrintPageEventArgs e)
{
 int x = 20;
 int y = 20;

 while (linesPrinted < lines.Length)
 {
  e.Graphics.DrawString (lines[linesPrinted++],
   new Font("Arial", 10), Brushes.Black, x, y);
  y += 15;
  if (y >= e.PageBounds.Height - 80)
  {
   e.HasMorePages = true;
   return;
  }
 }

 linesPrinted = 0;
 e.HasMorePages = false;
}
```

3. Dodajte rutinu za obradu događaja **Beginprint** objekta printDocument. Rutina **OnBeginPrint()** se poziva jednom za svaki zadatak štampe, i ovde pravimo naš niz **lines**:

```
private void OnBeginPrint(object sender, System.Drawing.Printing.PrintEventArgs e)
{
  char[] param = {'\n'};
  lines = txtBoxEdit.SelectedText.Split(param);

  int i = 0;
  char[] trimParam = {'\r'};
  foreach (string s in lines)
  {
    lines[i++] = s.TrimEnd(trimParam);
  }
}
```

4. Posle izgradnje projekta možete pokrenuti zadatak za štampu dokumenta sa vise strana.

Kako to radi
Počinjanje zadatka štampe metodom **Print()** objekta PrintDocument za uzvrat poziva rutinu **BeginPrint()** jednom i **OnPrintPage()** za svaku stranu.

U rutini **BeginPrint()**, podelimo tekst iz okvira za tekst u niz stringova. Svaki string u nizu predstavlja jednu liniju, jer je delimo sa znacima nove linije (\n) i sklanjamo znak povratka na početak reda (\r), kao što smo radili i ranije:

```
char[] param = {'\n'};
lines = txtBoxEdit.SelectedText.Split(param);

int i = 0;
char[] trimParam = {'\r'};
foreach (string s in lines)
{
  lines[i++] = s.TrimEnd(trimParam);
}
```

Rutina OnPrintPage() se poziva posle BeginPrint(). Želimo da nastavimo štampanje dokle god je broj odštampanih linija manji od broja linija koliko treba da odštampamo. Svojstvo **lines.Length** vraća broj stringova u nizu lines. Promenljiva **linesPrinted** se povećava sa svakom linijom koju pošaljemo štampaču:

```
while (linesPrinted < lines.Length)
{
  e.Graphics.DrawString (lines[linesPrinted++],
    new Font("Arial", 10), Brushes.Black, x, y);
```

Posle štampanja linije, proveravamo da li je novo izračunata vertikalna pozicija van okvira strane. Dodatno smanjujemo okvire za 80 piksela, jer zapravo ne želimo da štampamo do samog kraja papira; mnogi štampači ovo ionako ne mogu da urade. Ako je ova pozicija dostignuta, svojstvo **HasMorePages** klase **PrintPageEventArgs** je podešeno na true, da bi kontroler znao da se metod OnPrintPage() mora pozvati još jednom. Zapamtite da objekat PrintController ima metod **PrintLoop()**, koji ima sekvencu za svaku stranu koju štampa. **PrintLoop()** će se zaustaviti ako svojstvo **HasMorePages** ima vrednost false. Predefinisana vrednost svojstva **HasMorePages** je **false**, tako da se štampanje zaustavlja:

```
    y += 15;
    if (y >= e.PageBounds.Height - 80)
    {
      e.HasMorePages = true;
      return;
    }
  }
```

Podešavanje strane

Margine strane do sada su fiksirane u programu. Promenimo aplikaciju da dozvoli korisniku da postavi margine na strani. Da bi ovo bilo moguće, dostupna je još jedna klasa dijaloga: **pageSetupDialog**.

Ova klasa nam omogućuje da konfigurišemo veličinu i izvor papira, orijentaciju i margine papira, a pošto te opcije zavise od štampača, izbor štampača može da se izvrši i iz ovog dijaloga.

Papir

Vrednost true za svojstvo **AllowPaper** znači da korisnik može da bira veličinu i izvor papira. Svojstvo **PageSetupDialog.PageSettings.PaperSize** vraća instancu klase **PaperSize** gde možemo da čitamo visinu, širinu i ime papira, sa svojstvima **Height**, **Width** i **PaperName**. PaperName navodi imena kao **Letter** i **A4**. Svojstvo **Kind** vraća enumeraciju gde možemo da dobijemo vrednost iz enumeracije **PaperKind**. Ovo može da bude jedna od tri vrednosti: **European**, **American** ili **Japanese**.

Svojstvo **PageSetupDialog.PageSettings.Papersource** vraća instancu klase **PaperSource** gde možemo da pročitamo ime izvora i tipa papira (kao i to da li je štampač ispravno konfigurisan sa podešavanjima za štampač).

Margine

Postavljanje svojstva **AllowMargins** na true dozvoljava korisniku da postavi vrednost margine za štampu. Možemo definisati minimalne vrednosti za korisnikov unos, podešavajući svojstvo MinMargins.Da bismo pročitali margine, koristimo svojstvo **PageSetupDialog.PageSettings.Margins**. Vraćeni objekat Margins ima svojstva **Bottom**, **Left**, **Right** i **Top**.

Orijentacija papira

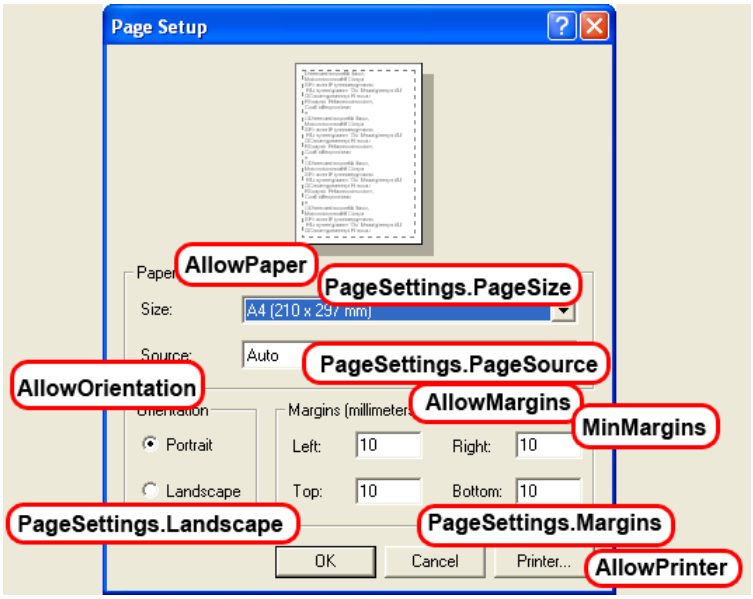
Svojstvo **AllowOrientation** definiše da li korisnik može da bira orijentaciju papira (**portrait** ili **landscape**). Izabrana vrednost se može pročitati iz svojstva **PageSetupDialog.PageSettings.Landscape**, što daje vrednost tipa **Boolean** za naglašavanje **landscape** režima sa **true** i **portrait** režima sa **false**.

Štampač

Svojstvo **AllowPrinter** definiše da li korisnik može da bira štampač. Zavisno od vrednosti ovoga svojstva, dugme **Printer** je vidljivo ili ne. Rutina za obradu pritiska ovog dugmeta otvara dijalog **PrintDialog** koji ćemo koristiti dalje.

Svojstva dijaloga PageSetupDialog

Slika ispod nam daje pregled svojstava koja uključuju ili isključuju posebne opcije ovog dijaloga, kao i koja svojstva mogu da se koriste za pristup ovim vrednostima:



Dodavanje dijaloga PageSetupDialog

1. Dodajte komponentu PageSetupDialog iz okvira sa alatima u dizajneru Windows formulara. Postavite svojstvo **Name** na **dlgPageSetup** i **Document** na **PrintDocument**, kako biste povezali dijalog sa dokumentom za štampu.

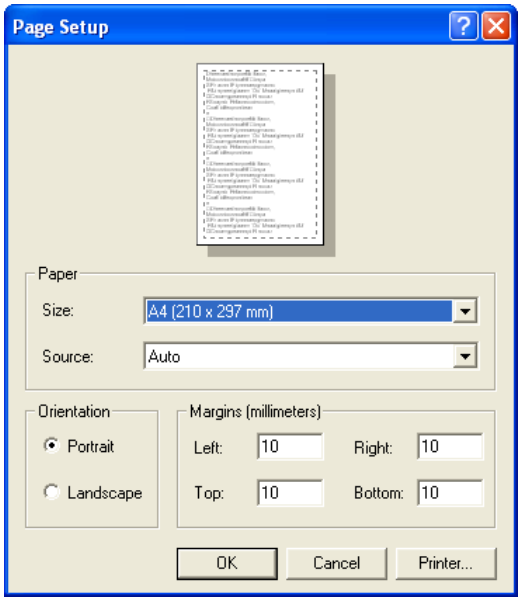
Tada dodajte rutinu za obradu događaja **Click** stavke u meniju **Page Setup**, i dodajte kod za prikaz dijaloga koristeći metod **ShowDialog()**. Nije neophodno proveravati povratnu vrednost metoda **ShowDialog()** ovde, jer implementacija rutine za obradu događaja **Click** dugmeta OK već postavlja nove vrednosti u povezanom objektu **PrintDocument**:

```
private void miFilePageSetup_Click(object sender, System.EventArgs e)
{
        dlgPageSetup.ShowDialog();
}
```

2. Sada promenite implementaciju metoda **OnPrintPage()** da koristi margine koje su postavljene u dijalogu **PageSetupDialog**. U našem kodu, promenljive **x** i **y** su postavljene na svojstva **MarginBounds.Left** i **MarginBounds.Bottom** klase **PrintPageEventArgs**. Proveravamo margine pomoću svojstva **MarginBounds.Bottom**:

```
private void OnPrintPage(object sender, System.Drawing.Printing.PrintPageEventArgs e)
{
 while (linesPrinted < lines.Length)
 {
  e.Graphics.DrawString (lines[linesPrinted++],
    new Font("Arial", 10), Brushes.Black, x, y);

  y += 15;
  {
   e.HasMorePages = true;
   return;
  }
 }

 linesPrinted = 0;
 e.HasMorePages = false;
}
```

3. Sada možete izgraditi projekt i pokrenuti aplikaciju. Izborom **File | Page Setup** pokazuje se sledeći dijalog. Možete promeniti margine i štampati sa podešenim okvirima strane:



*Ako prikazivanje dijaloga **PageSetupDialog** ne prode, izuzetak tipa **System.ArgumentException** se generiše; to je verovatno zato što ste zaboravili da povežete objekat **PrintDocument** sa kontrolom **PageSetupDialog**. Dijalogu **PageSetupDialog** treba povezani objekat **PrintDocument** da upita i podesi vrednosti koje su prikazane u dijalogu.*

Klasa PrintDialog

Klasa PrintDialog dozvoljava korisniku da izabere željeni štampač od instaliranih štampača, da izabere broj kopija, i neka podešavanja štampača kao što su izgled i izvor papira. Pošto je klasa PrintDialog veoma jednostavna za korišćenje, počećemo odmah sa dodavanjem dijaloga PrintDialog našoj aplikaciji.

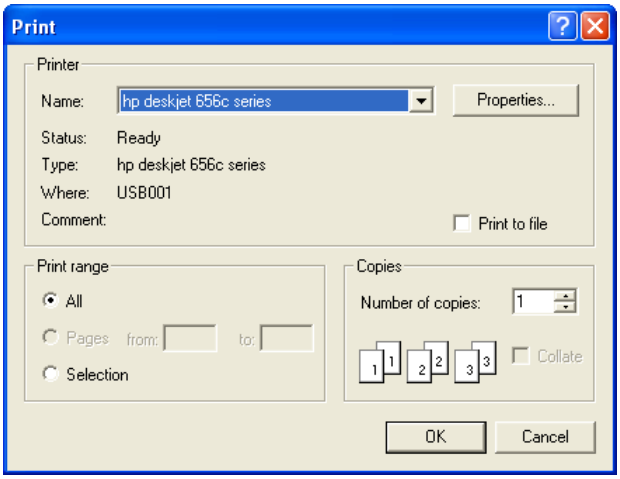
Dodavanje dijaloga PrintDialog

1. Dodajte komponentu **PrintDialog** iz okvira sa alatima na formular. Postavite svojstvo **Name** na **dlgPrint** i **Document** na **PrintDocument**.

Promenite implementaciju rutine za obradu događaja **Click** stavke menija **Print**, na sledeći način:

```
private void miFilePrint_Click(object sender, System.EventArgs e)
{
  if (txtBoxEdit.SelectedText != "")
  {
    printDocument.Print();
  }

}
```

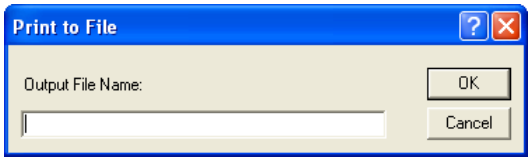
2. Izgradite i pokrenite aplikaciju. Izborom **File | Print** otvara se **PrintDialog**. Sada možete izabrati štampač za štampu dokumenta:



Opcije za PrintDialog

U našem programu Prosti Editor, nismo menjali nijedno svojstvo dijaloga PrintDialog. Ovaj dijalog ima takođe neke opcije. U gornjem dijalogu, možete videti tri grupe: **Printer**, **Print range** i **Copies**:

- U grupi Printer ne morate izabrati samo štampač; postoji i opcija **Print to File**. Ova opcija je podrazumevano omogućena, ali ne i potvrđena. Izborom ovog polja za potvrdu omogućuje se štampa u datoteku, umesto na štampač. Možete onemogućiti ovu opciju postavljanjem svojstva **AllowPrintToFile** na **false** . Ako korisnik izabere ovu opciju, sledeći dijalog se otvara pozivom **printDocument.Print()** tražeći ime datoteke u koju će se štampati.



- U sekciji dijaloga **Print Range** samo **All** može da bude izabran - opcije **Pages i Selection** su podrazumevano neaktivne. Pogledaćemo kako ove opcije mogu da budu implementirane u sledećoj sekciji.
- Grupa **Copies** dozvoljava korisniku da izabere broj kopija za štampu.

Štampanje izabranog teksta

Ako želite da omogućite da se štampa samo izabrani tekst, možete da aktivirate ovo radio dugme postavljanjem svojstva **AllowSelection** na vrednost **true** . Takođe morate da promenite kod za štampu tako da samo izabrani tekst biva odštampan.

Dodavanje štampanja izabranog teksta

1. Dodajte naglašeni kod u rutinu za obradu događaja Click dugmeta Print:

```
private void miFilePrint_Click(object sender, System.EventArgs e)
{
}
```

U našem programu, sve linije koje će biti odštampane postavljene su u rutini **OnBeginPrint()**. Promenite implementaciju ove metode:

```
private void OnBeginPrint(object sender, System.Drawing.Printing.PrintEventArgs e)
{
 char[] param = {'\n'};

   lines = txtBoxEdit.Text.Split(param);

 int i = 0;
 char[] trimParam = {'\r'};
 foreach (string s in lines)
 {
  lines[i++] = s.TrimEnd(trimParam);
 }
}
```

Sada možete izgraditi i pokrenuti program. Otvorite datoteku, izaberite neki tekst, pokrenite dijalog za štampanjem iz menija **File |Print**, i izaberite opciono dugme Selection iz grupe **Print Range**. Pošto je ovo izabrano, pritiskanje dugmeta **Print** će da odštampa samo izabrani tekst.

Kako to radi

Postavili smo svojstvo **AllowSection** na **true** samo ako je neki tekst izabran. Pre nego što smo prikazali **PrintDialog**, moramo da proverimo da li je tekst izabran. To se čini proverom da li je vrednost svojstva **SelectedText** okvira za uređivanje različita od **null**. Ako je nešto teksta izabrano, svojstvo **AllowSelection** je postavljeno na **true**:

```
if (txtBoxEdit.SelectedText != "")
{
  dlgPrint.AllowSelection = true;
}
```

OnBeginPrint() se poziva na početku svakog zadatka za štampu. Pristupom svojstvu **printDialog.PrinterSettings.PrintRange**, dobijamo informaciju da li je korisnik izabrao opciju Selection. Svojstvo **PrintRange** vraća enumeraciju **PrintRange** koja može da ima vrednosti **AllPages**, **Selection**, ili **SomePages**:

```
if (dlgPrint.PrinterSettings.PrintRange == PrintRange.Selection)
```

Ako je opcija **Selection**, dobijamo izabrani tekst string u svojstvu **SelectedText** kontrole **TextBox**. Ovaj string je podeljen na isti način kao i kompletan tekst:

```
lines = txtBoxEdit.SelectedText.Split(param);
```

Za korišćenje enumeracije **PrintRange** moramo da uvrstimo imenovani prostor **System.Drawing.Printing** na početku programa:

```
using System.Drawing.Printing;
```

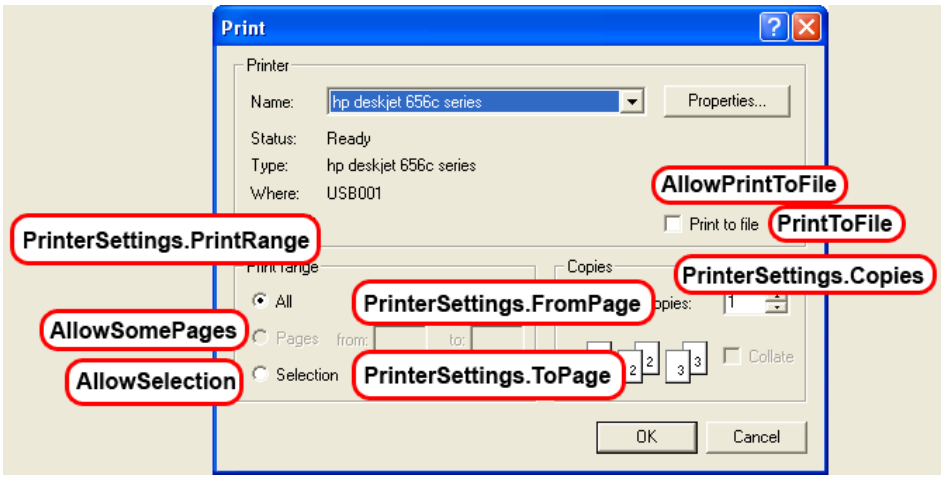
Štampanje opsega strana

Štampanje opsega strana može biti implementirano na sličan način kao i štampanje izbora. Opciono dugme može biti aktivirano podešavanjem svojstva **AllowSomePages** na **true**. Korisnik sada može da bira od koje do koje stranice želi da štampa. Ipak, gde su granice stranica u našem jednostavnom programu za uređivanje? Koja je zadnja stranica? Trebalo bi da podesimo zadnju stranicu izmenama u svojstvu **PrintDialog.PrinterSettings.ToPage**.

Kako korisnik zna brojeve stranica koje želi da odštampa? To nije problem u aplikacijama kao što je Microsoft Word, gde **Print Layout** može da bude izabran kao pogled ekrana. Ovo nije moguće sa jednostavnom kontrolom TextBox koju koristimo u našem Prostom Editoru. To je razlog zašto nećemo implementirati ovu mogućnost u našu aplikaciju.
Naravno, ovo možete uraditi kao vežbu. Svojstvo **AllowSomePages** mora biti postavljeno na true. Pre prikazivanja dijaloga **PrintDialog**, možete takođe da postavite svojstvo **PrinterSettings.ToPage** na maksimalan broj strana.

Svojstva dijaloga PrintDialog

Da sumiramo svojstva koja utiču na izgled dijaloga PrintDialog ponovo sa slikom:



Pregled pre štampanja

Možda korisnik želi da vidi kako će izgledati štampana strana. To je razlog zašto ćemo iskoristiti pregled pre štampanja. Implementiranje pregleda pre štampanja u .NET okruženju je lako. Možemo upotrebiti klasu **PrintPreviewControl** koja se koristi da uradi pregled dokumenta u formularu, da pokaže kako će biti odštampana. **PrintPreviewDialog** je dijalog koji obmotava kontrolu.

Dijalog PrintPreviewDialog

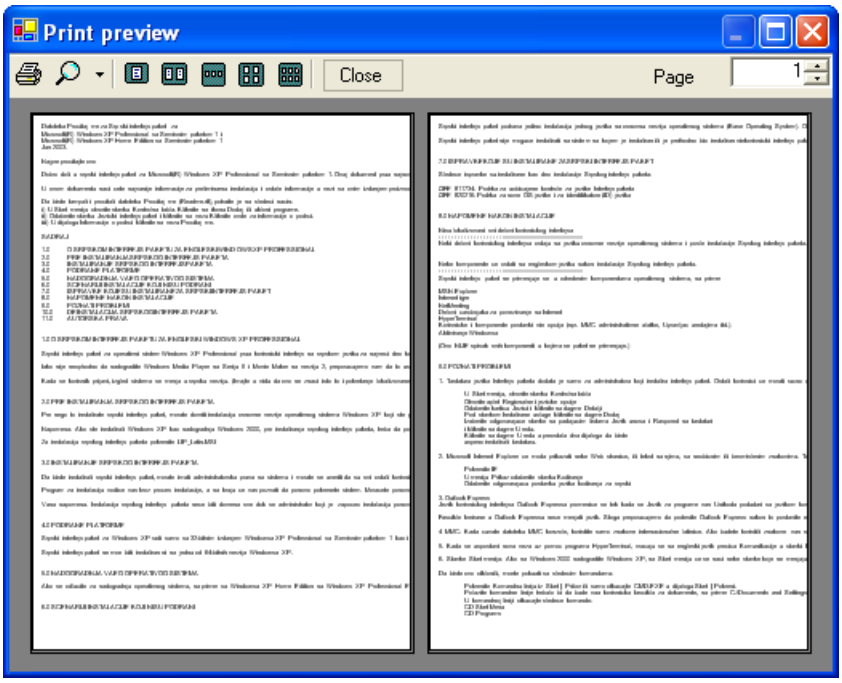
Posmatrajući svojstva i listu nasleđa iz MSDN dokumentacije za klasu **PrintPreviewDialog**, možete lako videti da je ovo formular, a ne obmotani opšti dijalog - klasa se razlikuje od **System.WindowsForms.Form**, i možete da radite sa njom sa formularima - klasa je izvedena iz **System.Windows.Forms.Form**, i možete da radite sa njom kao sa formularima. Dodaćemo klasu **PrintPreviewDialog** u našu aplikaciju Prosti Editor.

Dodavanje dijaloga Print Preview

1. Dodajte komponentu **PrintPreviewDialog** u dizajneru Windows formulara. Postavite svojstvo Name na **dlgPrintPreview**, i **Document** na **PrintDocument**.

Dodajte i implementirajte rutinu za obradu događaja **Click** stavke menija **Print Preview**:

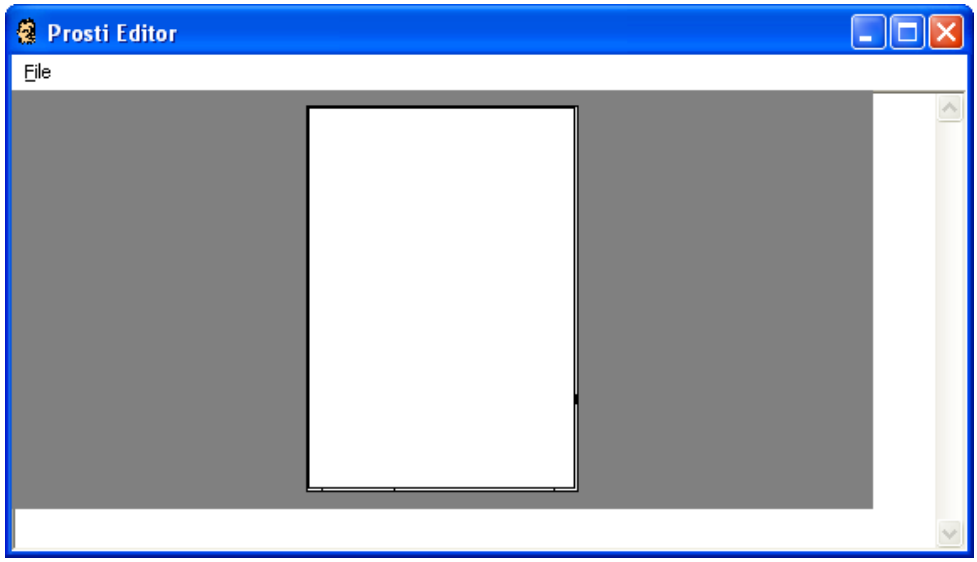
```
private void miFilePrintPreview_Click(object sender, System.EventArgs e)
{
   dlgPrintPreview.ShowDialog();
}
```



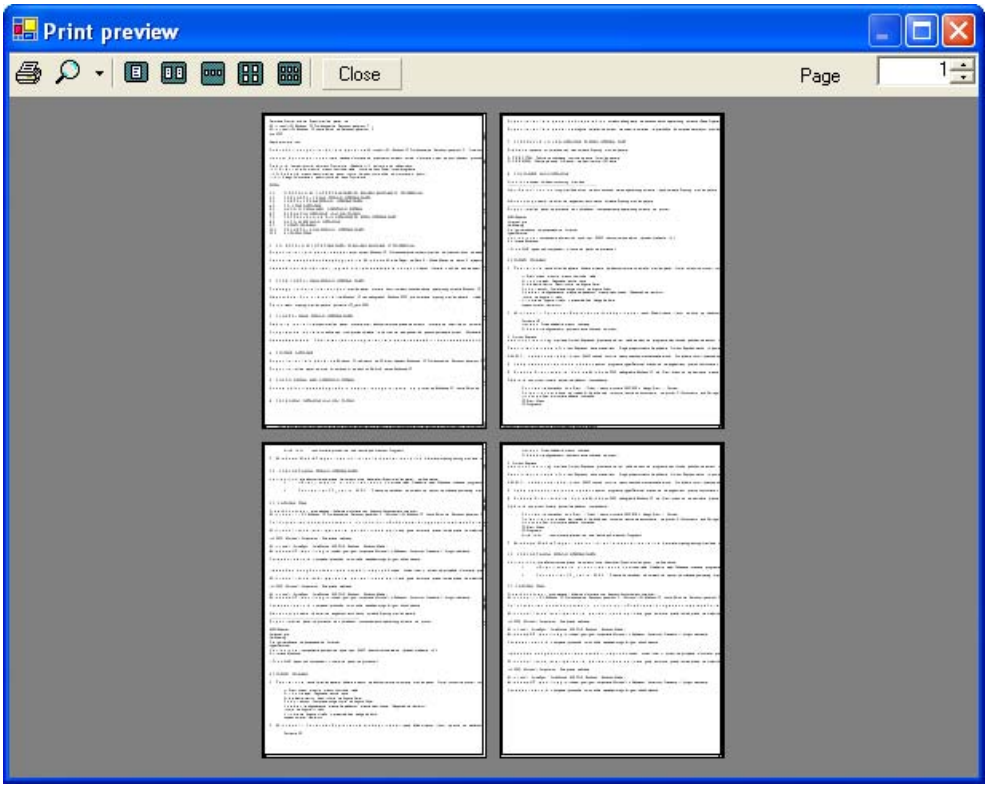
Klasa PrintPreviewControl

Pregled pre štampanja u Microsoft Wordu i WordPadu je drugačiji od dijaloga **PrintPreviewDialog** u tome što se u ovim aplikacijama pregled ne pojavljuje u sopstvenom dijalogu, nego u glavnom prozoru aplikacije.

Da bi se izvelo isto, možete da postavite klasu **PrintPreviewControl** u vaš formular. Svojstvo **Document** mora da bude postavljeno na objekat **PrintDocument**, a **Visible** na **false**. U rutini možete postaviti svojstvo **Visible** na **true** kako biste videli pregled pre štampanja. Tada je **PrintPreviewControl** ispred drugih kontrola, kao što je pokazano na sledećoj slici:



Iz naslova i jedne jedine stavke u meniju File možete da vidite da je prikazan glavni prozor aplikacije Prosti Editor. Treba još da dodamo neke elemente koji kontrolišu klasu **PrintPreviewControl** da vrši zumiranje, štampu, i prikazivanje teksta u 2, 3, 4 ili 6 strana, itd. Posebna paleta sa alatkama može da se koristi da učini ove mogućnosti dostupnim. Klasa **PrintPreviewDialog** već ovo ima implementirano, kao što možete da vidite u sledećoj slici.



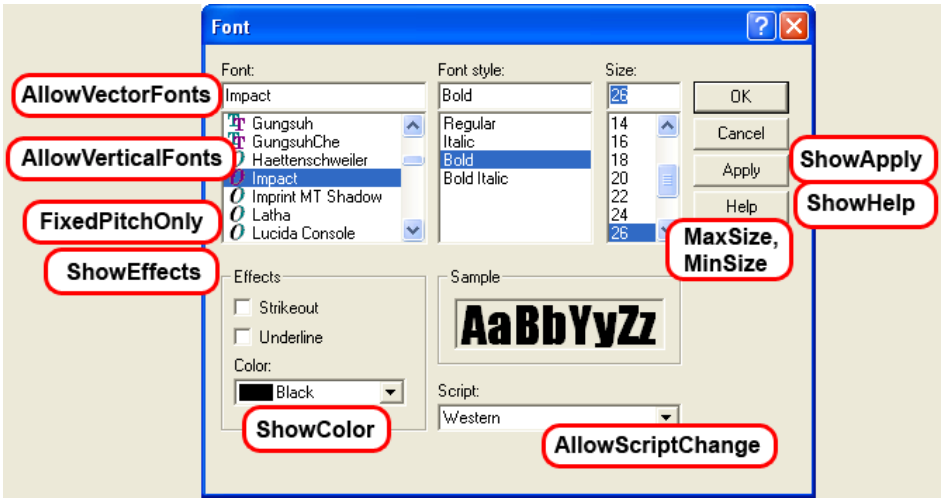
Klase FontDialog i ColorDialog

Poslednji dijalozi o kojima ćemo govoriti u ovom predavanju su **FontDialog** i **ColorDialog**.
Ovde ćemo se zadržati na diskusiji dijaloga zapostavljanje fonta i boje, a ne i na klasama **Font** i **Color**.

Klasa FontDialog

Klasa FontDialog omogućuje korisniku aplikacije izbor fonta. Korisnik može da menja font, stil,veličinu i boju.
Sledeća slika bi trebalo da vam omogući pregled svojstava koji menjaju elemente u dijalogu:

Klasa FontDialog omogućuje korisniku aplikacije izbor fonta. Korisnik može da menja font, stil, veličinu i boju.



Kako se koristi klasa FontDialog

Dijalog može da se koristi na isti način kao i u prethodnim dijalozima. U dizajneru Windows formulara, dijalog može biti prevučen iz okvira sa alatima i spušten na formular tako da se kreira instanca klase **FontDialog**.
Kod koji koristi **FontDialog** izgleda ovako:

```
if (dlgFont.ShowDialog()==DialogResult.OK)
{
  txtBoxEdit.Font=dlgFont.Font;
}
```

Dijalog **FontDialog** je prikazan pozivanjem metode **ShowDialog()**. Ako korisnik pritisne dugme OK, metod vraća **DialogResult.OK**. Izabrani font može da se čita koristeći svojstvo Font klase **FontDialog**; ovaj font se tada prosleđuje svojstvu **Font** kontrole **TextBox**.

Svojstva klase FontDialog

Već smo videli sliku sa svojstvima klase **FontDialog**, a možemo da vidimo i za šta se ova svojstva koriste:

Svojstvo	Opis
AllowVectorFonts	Boolean vrednost koja definiše da li vektorski fontovi mogu da budu izabrani u listi fontova.
AllowVerticalFonts	Boolean vrednost koja definiše da li se vertikalni fontovi mogu izabrati u listi fontova. Vertikalni tekstovi se koriste u zemljama dalekog istoka. Verovatno nemate vertikalni font instaliran u vašem sistemu.
FixedPitchOnly	Postavljanje svojstva FixedPitchOnly prikazuje samo fontove fiksirane širine. Sa fontom fiksirane širine svaki znak ima istu veličinu. Podrazumevano je false .
MaxSize	Definisanje vrednosti za MaxSize svojstvo definiše maksimalnu veličinu fonta koju korisnik može da izabere.
MinSize	Slično kao i MaxSi ze, možete postaviti najmanju veličinu fonta koju korisnik može izabrati.

ShowApply	Ako dugme Apply treba da bude prikazano, morate da postavite svojstvo ShowApply na true. Pritiskanjem dugmeta Apply, korisnik može da vidi promenu fonta u aplikaciji bez napuštanja tekst dijaloga.
ShowColor	Izbor boje nije podrazumevano prikazan u dijalogu. Ako želite da korisnik izabere boju fonta u font dijalogu, morate da postavite svojstvo ShowColor na true.
ShowEffects	Korisnik može podrazumevano da izabere polja za potvrdu Strikeout i Underline da upravlja fontom. Ako ne želite da ove opcije budu prikazane, morate da postavite ShowEffects na false.
AllowScriptChange	Postavljanje svojstva AllowScriptChange na false sprečava korisnika da promeni pisma fonta. Dostupna pisma zavise od izabranog fonta, na primer: font Arial podržava zapadno, hebrejsko, arapsko, grčko, tursko, baltičko, centalnoevropsko, ćirilično i vijetnamsko pismo.

Uključivanje dugmeta Apply

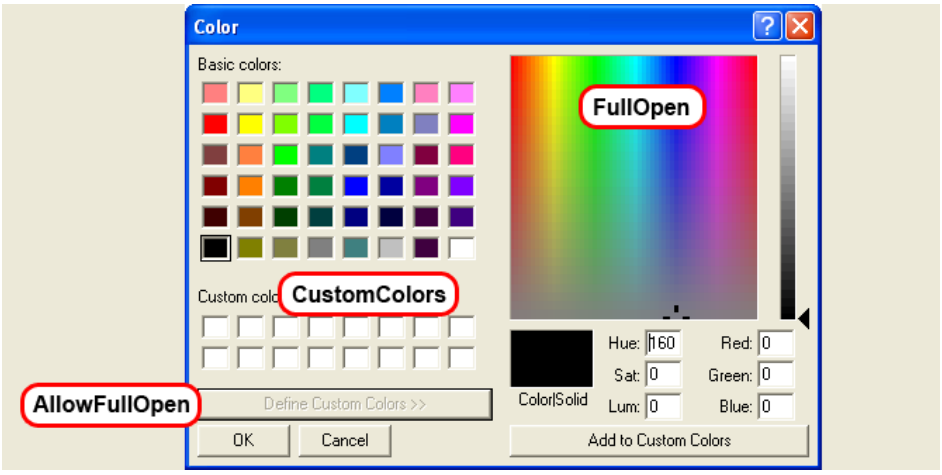
Zanimljiva razlika u odnosu na druge prikazane dijaloge jeste što **FontDialog** podržava dugme **Apply** koje podrazumevano nije prikazano. Ako korisnik pritisne dugme **Apply**, dijalog ostaje otvoren, ali izbor fonta treba da bude primenjen nad tekstom.

Birajući komponentu FontDialog u dizajneru Windows formulara, možete postaviti svojstvo **ShowApply** u prozoru Properties na true. Ali, kako smo obavešteni da li korisnik sad pritiska dugme Apply? Dijalog je još otvoren, tako da se metod **ShowDialog()** još neće završiti. Umesto toga, možemo dodati rutinu za obradu događaja Apply klase FontDialog. Ovo možete učiniti pritiskajući dugme Events u prozoru Properties, i upisivanjem imena rutine za događaj Apply.
Kao što možete videti u sledećem kodu, uneto je ime **OnApplyFontDialog**. U ovoj rutini možete pristupiti izabranom fontu u dijalogu FontDialog koristeći promenljivu članicu klase FontDialog:

```
private void miColor_Click(object sender, System.EventArgs e)
{
 if (dlgColor.ShowDialog()==DialogResult.OK)
 {
  txtBoxEdit.ForeColor=dlgColor.Color;
 }
}
```

Klasa ColorDialog

Klasa FontDialog nema ni približno tako mnogo svojstava za podešavanje kao ColorDialog. Sa klasom ColorDialog korisniku je moguće da podesi posebne boje, ako ne želi ni jednu od ponuđenih osnovnih boja; ovo se radi podešavanjem svojstva **AllowFullColor**. Deo sa konfigurisanjem posebnih boja u dijalogu može biti automatski proširen svojstvom **FullColor**. Svojstvo **SolidColorOnly** definiše da se mogu koristiti samo osnovne boje. Svojstvo **CustomColors** može se koristiti za dobijanje i konfigurisanje proizvoljnih boja:



Kako se koristi dijalog ColorDialog

Komponenta **ColorDialog** može biti prevučena iz okvira sa alatima i spuštena na formularu dizajneru Windows formulara, kao što smo radili već sa drugim dijalozima. Metod **ShowDialog()** prikazuje dijalog dok korisnik ne pritisne dugme **OK** ili **Cancel**. Možete pročitati izabranu boju pristupanjem svojstvu Color dijaloga, kao što se vidi u sledećem primeru koda:

```
if (dlgColor.ShowDialog() == DialogResult.OK
 {
   txtboxEdit.ForeColor = dlgColor.Color;
  }
```

Svojstva klase ColorDialog

Svojstva koja utiču na izgled dijaloga su prikazana u ovoj tabeli:

Svojstvo	Opis
AllowFullOpen	Podešavanje ovog svojstva na false ne dozvoljava korisniku da definiše posebnu boju, jer je dugme **Define Custom Colors** deaktivirano. Podrazumevana vrednost ovog svojstva je true.
FullOpen	Podešavanje svojstva **FullOpen** na true pre nego što se dijalog prikaže, automatski otvara dijalog sa posebnim podešavanjima boja.
AnyColor	Postavljanjem ovog svojstva na true pokazuju nam se sve dostupne boje u listi osnovnih boja.
CustomColors	Sa svojstvom **CustomColors** možete podesiti niz posebnih boja, i možete pročitati posebne boje koje je korisnik podesio.
SolidColorOnly	Podešavanjem svojstva **SolidColorOnly** na true, korisnik može samo da izabere osnovne boje.

Korišćenje kontrola za Windows formulare - I deo

Poslednjih godina Visual Basic pružao je programerima alate za kreiranje detaljnih korisničkih interfejsa, kroz intuitivni **dizajner formulara.** Ovaj dizajner sa svojim alatima, zajedno sa programskim jezikom lakim za učenje, predstavljao je možda najbolje okruženje za razvoj velikih aplikacija. Jedna od stvari koje Visual Basic radi, kao i neki dragi alati za razvoj aplikacija (npr. Delphi), jeste obezbeđivanje pristupa predefinisanim kontrolama, koje programer može brzo ugraditi u korisnički interfejs aplikacije.

U centra većine Visual Basic Windows aplikacija nalazi se dizajner formulara. Korisnički interfejs se kreira tako što se kontrole prevlače iz okvira za alate i spuštaju na obrazac tamo gde želimo da stoje kada se program izvršava. Dvostrukim pritiskom na kontrole dodaju se **ratine za obradu** kontrola. Microsoftove unapred definisane kontrole, zajedno sa standardnim kontrolama, koje se mogu kupiti po razumnim cenama, predstavljaju nov izvor potpuno testiranog koda, koji se može uvek koristiti. Ono što je bilo najvažnije u Visual Basicu, sada je, kroz Visual Studio .NET, dostupno C# programerima.

Većina kontrola korišćenih pre .NET-a bile su, i još uvek jesu, specijalni COM objekti, poznatiji kao ActiveX kontrole. Ove kontrole su se prikazivale i za vreme dizajniranja i za vreme izvršavanja programa. Svaka kontrola ima određena svojstva, dopuštajući programeru izvestan nivo prilagođavanja, kao što je podešavanje boje pozadine, naslova i njegove pozicije u formulara. Kontrole koje ćemo videti u ovom predavanju izgledaju isto kao i ActiveX kontrole, ali nisu - to su NET sklopovi. Ipak, i dalje se mogu koristiti kontrole dizajnirane za starije verzije Visual Studija, iako produžavaju rad zato što je potrebna dodatna obrada ovih kontrola od strane .NET-a. Iz očiglednih razloga, kada je .NET dizajniran, Microsoft nije hteo da napravi suvišnim postojeće kontrole, već je pronašao način da se i one upotrebe i pored toga što će sve buduće kontrole biti iste .NET komponente.

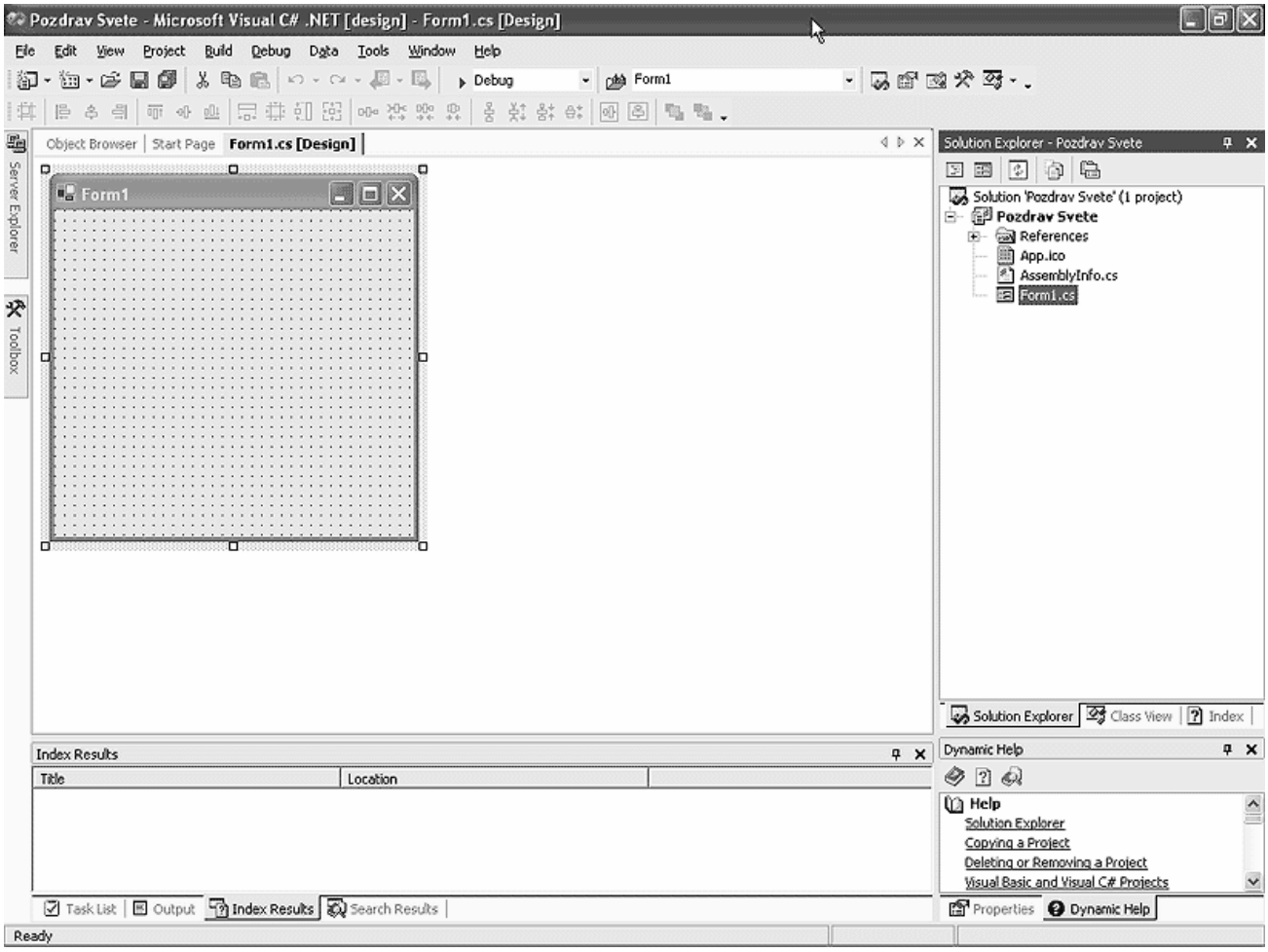
Ovi .NET sklopovi mogu biti dizajnirani tako da se koriste u bilo kojem od jezika Visual Studio. U budućnosti, veruje se, proizvodiće se samo čiste .NET komponente.

Već smo imali prilike da vidimo dizajner formulara, nakratko, u ovoj knjizi. U ovom predavanju detaljnije ćemo razmatrati dizajner, kao i izvestan broj kontrola iz okvira Visual Studio .NET-a. Predstaviti sve kontrole trenutno prisutne u Visual Studio .NET-u, bilo bi isuviše za ovu knjigu; zato ćemo predstaviti samo one najčešće korišćene, počevši od oznaka i okvira za tekst, do prikaza lista i statusnih linija.

Dizajner Windows formulara

Počećemo kratkim pregledom Windows dizajnera formulara. Ovo je centralno mesto za kreiranje korisničkog interfejsa. Potpuno je moguće kreirati formulare bez korišćenja Visual Studio .NET-a; međutim, dizajniranje interfejsa u Notepadu može biti težak posao.

Da počnemo od okruženja u kojem ćemo raditi. Pokrenite Visual Studio .NET i kreirajte novi C# Windows Application projekat tako što ćete izabrati File | New | Project. U okviru za dijalog koji se pojavljuje, u stablu sa leve strane pritisnite Visual C# Projects, a zatim izaberite Windows Application u listi sa desne strane. 2a sada ne menjajte ime koje je predložio Visual Studio i pritisnite OK. To bi trebalo da kreira prozor sličan ovome ispod:



Ako ste upoznati sa dizajnerom formulara iz Visual Basica, primetićete određene sličnosti - dizajner se pokazao kao dobro oruđe i neko je odlučio da ga koriste i drugi jezici Visual Studija. Ako niste upoznati sa dizajnerom, razmotrimo šta se sve pojavljuje na ovom prozoru, panel po panel.

U centru samog prozora nalazi se formular koji dizajniramo. Kontrole se mogu prevlačiti iz okvira sa alatima i spuštati na formular. Okvir sa alatima se trenutno ne vidi na slici iznad. Ali, al pomerite pokazivač miša skroz ulevo, iznad jezička kartice Toolbox, pojaviće se. Možete da pritisnete pribadaču u gornjem desnom uglu panela da ga „otvorite". Ovo će prearanžirati izgled radnog prostora, tako da se okvir sa alatima uvek vidi i neće smetati samoj formi. Uskoro ćemo pogledati sam okvir sa alatima i šta sve on sadrži.

Takođe skriven, na levoj traci, nalazi se i Server Explorera - predstavljen ikonama na kojoj su nacrtani računari, iznad jezička kartice Toolbox. Ovo se može posmatrati i kao kraća verzija Windows Control Panela. Odavde možete pretraživati mrežu, dodavati ili uklanjati veze sa bazama podataka i još mnogo toga drugog.

Sa desne strane prozora nalaze se dva panela. Gornji je Solution Explorer i pregled klasa. U Solution Exploreru možete videti sve otvorene projekte i njihove pridružene datoteke. Ako pritisnete na karticu pri dnu Solution Explorera, aktivirate Class Viewer (pregled klasa). Ovde možete pregledati sve klase korišćene u projektima, kao i sve izvedene klase.

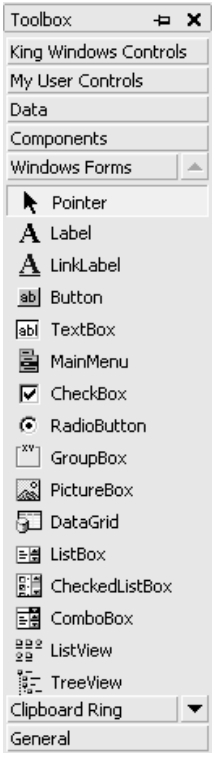
U donjem desnom delu prozora nalazi se panel Properties. Ovaj panel sadrži sva svojstva izabranog objekta, koja se lako mogu pročitati ili izmeniti, tj. dopuniti. Cesto ćemo koristiti ovaj panel u ovom predavanju.

Takođe, u ovom panelu vidi se i kartica Dynamic Help. Ovde se prikazuju svi saveti i uputstva za izabrani objekat ili kod, čak i dok kucate. Ako imate stariji računar sa malom količinom radne memorije, predlažemo da uklonite ovaj panel kada on nije potreban, zato što aktivna potraga saveta i pomoći može usporiti rad računara.

Okvir sa alatima

Pogledajmo malo bolje okvir sa alatima - Toolbox. Ako već to niste uradili, pomerite pokazivač miša do okvira na levoj strani ekrana, i pritisnite **pin** u gornjem desnom uglu panela, tako da ceo panel bude vidljiv.

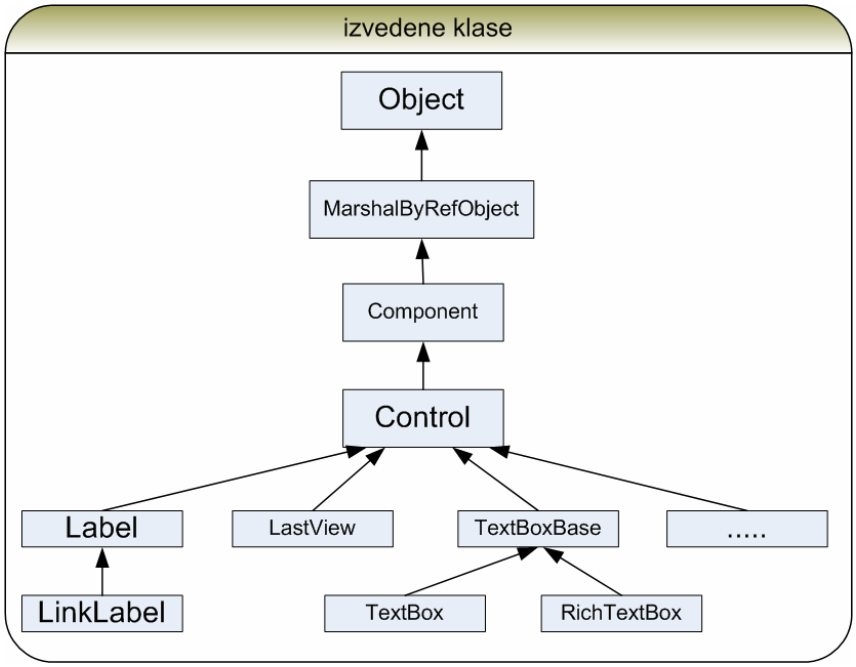
Ako slučajno pritisnete X umesto pribadače i tako uklonite okvir sa alatima, možete ga ponovo vratiti u prozor. Izaberite Toolbox iz menija View, ili pritisnite tastere Ctrl+Alt+X



Okvir sa alatima sadrži sve kontrole dostupne .NET programerima. Što je važnije, okvir sadrži sve kontrole za programere Windows aplikacija. Da ste izabrali kreiranje projekta Web Forms, umesto projekta Windows Application, dobili biste drugi okvir sa alatima. Niste obavezni da koristite ovakav izbor kontrola. Okvir sa alatima možete prilagoditi svojim potrebama, ali u ovom predavanju fokusiraćemo se na kontrole koje se nalaze u okviru (slika iznad). U stvari, pogledaćemo većinu od njih.

Kontrole

Većina kontrola u .NET-u izvedene su iz klase **System.Windows.Forms.Control**. Ova klasa definiše osnovno funkcionisanje kontrola, zbog čega su mnogi događaji i osobine kontrola identični. Mnoge od ovih klasa su osnovne klase za druge kontrole, kao što je to slučaj sa klasama Labeli TextBoxBase:



Neke kontrole, koje se nazivaju korisničkim kontrolama, izvedene su iz druge klase: System.Windows.Forms.UserControl. Ova klasa je izvedena iz klase Control i omogućuje nam da kreiramo sopstvene kontrole. Takođe, kontrole koje koristimo za dizajniranje Web korisničkog interfejsa izvedene su iz klase System.Web.UI.Control.

Osobine

Sve kontrole imaju određene osobine koje određuju kako će se neka kontrola ponašati. Osnova klasa za većinu kontrola, Control, sadrži određen broj svojstava koje kontrole ili direktno nasleđuju, ili redefinišu u slučaju da se želi neko drugačije ponašanje kontrole.

Tabela ispod pokazuje najvažnije osobine klase Control. Ove osobine biće prisutne u većini kontrola koje demo videti u ovom predavanju. *Zbog* toga ih nećemo ponovo detaljno objašnjavati, osim ukoliko ne menjamo ponašanje konkretne kontrole. Tabela nije napisana da zamori čitaoca; ukoliko želite da pronađete sve osobine klase Control, pogledajte MSDN biblioteku.

Ime osobine	Dostupnost	Opis
Anchor	Čitanje/upis	Ovo svojstvo definiše ponašanje kontrole kada kontejner koji sadrži kontrolu menja svoju veličinu. Posle tabele nalazi se detaljno objašnjenje ove osobine.
BackColor	Čitanje/upis	Boja pozadine kontrole.
Bottom	Čitanje/upis	Određuje razdaljinu od vrha prozora do dna kontrole. Ovo nije isto što i visina kontrole.
Dock	Čitanje/upis	Dopušta da kontrola bude pripojena uz ivicu prozora. Posle tabele nalazi se detaljno objašnjenje ove osobine.
Enabled	Čitanje/upis	Podesiti svojstvo Enabled na true (tačno) obično znači da kontrola može da primi unos od strane korisnika. Podešeno na false (netačno) obično znači da ne može.
ForeColor	Čitanje/upis	Boja prednjeg plana kontrole.
Height	Čitanje/upis	Udaljenost od vrha do dna kontrole.

Ime osobine	Dostupnost	Opis
Left	Čitanje/upis	Udaljenost leve ivice kontrole od leve ivice prozora.
Name	Čitanje/upis	Ime kontrole. Ime se koristi za pristup kontroli u kodu.
Parent	Čitanje/upis	Roditelj kontrole.
Right	Čitanje/upis	Udaljenost desne ivice kontrole od leve ivice prozora.
TabIndex	Čitanje/upis	Redosled promene kontrola u kontejneru koji ih drži prilikom pritiska na taster Tab.
Tabstop	Čitanje/upis	Određuje da li kontrola može biti dostupna pritiskanjem tastera Tab.
Tag	Čitanje/upis	Ovu vrednost ne koristi sama kontrola, već služi programerima za smeštanje informacija o kontroli. Ovoj osobini može se dodeliti jedino vrednost u formi stringa.
Top	Čitanje/upis	Udaljenost gornje ivice kontrole od vrha prozora.
Visible	Čitanje/upis	Određuje da li je kontrola vidljiva u toku izvršavanja programa.
Width	Čitanje/upis	Širina kontrole.

Svojstva Anchor i Dock

Ove dve osobine su naročito korisne kada dizajnirate obrazac. Osigurati da prozor normalno izgleda posle promene veličine prozora, nije nimalo lako. Ogroman broj redova koda napisan je da bi se ovo moglo izvesti. Mnogi programi rešavaju ovaj problem tako što ne dozvoljavaju da se menja veličina prozora, što je svakako najlakši način, ali ne i najbolji. Svojstva Anchor i Dock, koja su uvedena kroz .NET okruženje, omogućavaju vam da rešite ovaj problem bez pisanja koda,

Svojstvo Anchor određuje kako se kontrola ponaša kada korisnik promeni veličinu prozora u kojem se kontrola nalazi. Može se namestiti tako da i kontrola menja veličinu, zajedno sa prozorom, ili da ostane iste veličine i na istom odstojanju od ivice prozora.

Svojstvo Dock je povezano sa svojstvom Anchor. Određuje da li kontrola treba da bude pripojena uz ivicu prozora kojem pripada. Ukoliko korisnik promeni veličinu prozora, kontrola će i dalje biti pripojena uz ivicu. Ako, na primer, odredite da kontrola treba uvek da bude pripojena uz donju ivicu prozora, kontrola će menjati svoju veličinu i ostati u donjem delu, bez obzira na to kako menjate veličinu samog prozora.

Pogledajte primer okvira za tekst, kasnije u ovom predavanju, za tačnu primenu Anchor osobine,

Događaji

Kada korisnik programa pritisne dugme ili taster, programer bi hteo da zna da se to zaista desilo. Da bi ovo bilo moguće, kontrole koriste događaje. Klasa Control definiše brojne događaje koji su zajednički za kontrole. Tabela ispod opisuje neke od ovih događaja. Ovo je, kao i ranije, izbor najčešće korišćenih; ukoliko želite da pronađete celu listu događaja, pogledajte MSDN biblioteku.

Ime događaja	Opis
Click	Dešava se kada se pritisne na kontrolu. Ovaj događaj će se takođe desiti kada se pritisne taster Enter.
Doubleclick	Dešava se kada se kontrola dva puta pritisne. Obradom događaja nekim kontrolama, kao što je kontrola Button, znači da Doubleclick događaj nikad neće biti pozvan.
DragDrop	Dešava se kada se završi operacija prevlačenja i spuštanja. Drugim rečima, kada se neki objekat prevuče do kontrole, a onda se pusti taster miša.
DragEnter	Dešava se kada dovučeni objekat ude u granice kontrole.
DragLeave	Dešava se kada dovučeni objekat napušta granice kontrole.
DragOver	Dešava se kada dovučeni objekat prelazi preko kontrole.
KeyDown	Dešava se kada se taster pritisne dok kontrola ima fokus. Ovaj događaj uvek ide pre događaja KeyPress i KeyUp.
Key Press	Dešava se kada se taster pritisne dok kontrola ima fokus. Ovaj događaj uvek ide posle KeyDown, a pre KeyUp događaja. Razlika između KeyDown i KeyPress jeste ta da KeyDown prenosi kod tastature za taster koji je pritisnut, dok KeyPress prenosi odgovarajuću vrednost tipa char za taster.
KeyUp	Dešava se kada se taster pusti dok kontrola ima fokus. Ovaj događaj uvek ide posle događaja KeyDown i KeyPress.
GotFocus	Dešava se kada kontrola primi fokus. Nemojte koristiti ovaj događaj za validaciju kontrole - umesto toga koristite događaje Validating ili Validated.
LostFocus	Dešava se kada kontrola izgubi fokus. Nemojte koristiti ovaj događaj za validaciju kontrole - umesto toga koristite Validating ili Validated događaje.
MouseDown	Dešava se kada se pokazivač miša nalazi iznad kontrole i kada se taster miša pritisne. Ovo nije isto što i događaj Click; MouseDown se dešava čim se pritisne dugme i pre nego što se pusti.
MouseMove	Dešava se kontinuirano dok miš prelazi preko kontrole.
MouseUp	Dešava se kada je pokazivač miša na kontroli i kada se taster miša oslobodi.
Paint	Dešava se kada se kontrola iscrtava.
Validated	Ovaj događaj se dešava kada se kontrola sa CausesValidation osobinom podešenom na true, sprema da primi fokus. Počinje kada se završi događaj Validating i pokazuje da je validacija završena.
Validating	Ovaj događaj se dešava kada se kontrola sa CausesValidation osobinom podešenom na true, sprema da primi fokus. Kontrola nad čijim sadržajem se vrši validacija jeste ona kontrola koja gubi fokus, a ne ona koja ga prima.

Sada možemo da pređemo na same kontrole i počećemo sa jednom koju smo videli u prethodnim predavanju, kontrolom Button.

Kontrola Button

Kada se kaže dugme (Button), verovatno pomislite na četvorougaono dugme koje se može pritisnuti radi izvođenja neke radnje. Međutim, tehnički, postoje tri vrste dugmadi u Visual Studio .NET-u. To je zato što su radio-dugmad i **polja za potvrdu,** takođe dugmad. Zbog ovoga, klasa Button nije direktno izvedena iz klase Control, već iz klase ButtonBase, koja je izvedena iz Control. Fokusiraćemo se na kontrolu Button, a radio-dugme i polje za potvrdu ostaviti za kasnije.

Kontrola Button postoji na svakom okviru za dijalog kojeg možete da se setite. Button se, uglavnom, koristi za obavljanje tri vrste zadataka:

- Za zatvaranje okvira za dijalog (npr. OK ili Cancel dugme);
- Za izvršavanje neke akcije nad podacima unetim u okviru za dijalog (npr. pritisak na dugme **Search** posle unošenja nekog kriterijuma pretrage);
- Za otvaranje nekog drugog okvira za dijalog ili aplikacije (npr. dugme **Help).**

Rad sa kontrolom Button je veoma jednostavan. Obično se sastoji od stavljanja kontrole u formular, dvostrukog pritiska na kontrolu za dodavanje koda za događaj Click, što će bid dovoljno za većinu aplikacija sa kojima ćete radio.
Razmotrimo sada neke najčešće korišćene osobine i događaje kontrole. To će vam omogućiti da shvatite šta se sve može uraditi sa kontrolom. Zatim, napravićemo mali primer koji će demonstrirati neke od osnovnih svojstava i događaja dugmeta

Svojstva kontrole Button

Opisaćemo osobine kao članove klase Button, iako su, tehnički, ove osobine definisane u klasi ButtonBase. Samo one najčešće korišćene opisane su u tabeli. Za celu listu, pogledajte MSDN biblioteku.

Ime osobine	Dostupnost	Opis
FlatStyle	Čitanje/upis	Izgled dugmeta može biti izmenjen pomoću ove osobine. Ako namestite stil na **PopUp,** dugme će biti ravno sve dok se pokazivač miša ne pomeri na njega. Tada se dugme pojavljuje u normalnom, 3D obliku.
Enabled	Čitanje/upis	Pominjemo ovu osobinu ovde, iako je ona izvedena iz klase Control, zato što je od velike važnosti za dugme. Podešavanje Enabled osobine na false znači da dugme postaje sivo i ništa se ne dešava kada se pritisne na njega.
Image	Čitanje/upis	Omogućava biranje slike (bitmape, ikone itd.) koja će se pojaviti na dugmetu.
ImageAlign	Čitanje/upis	Uz pomoć ove osobine možete da izaberete položaj slike na dugmetu.

Događaji kontrole Button

Daleko najkorišćeniji događaj dugmeta jeste događaj Click. Dešava se kada korisnik pritisne na dugme; pod ovim se misli kada pritisne levo dugme miša i kada ga oslobodi dok se pokazivač miša nalazi na dugmetu. To znači da kada pritisnete levim dugmetom miša, a zatim pomerite pokazivač miša sa kontrole pre nego što oslobodite levo dugme miša, neće doći do događaja Click. Takođe, do događaja Click dolazi kada je dugme u fokusu i kada korisnik pritisne taster *Enter*. Ako imate dugme na formularu, uvek treba obraditi ovaj događaj.

Pređimo sada na primer. Kreiraćemo dijalog sa tri dugmeta. Dva će menjati korišćeni jezik sa engleskog na danski i obrnuto (možete da koristite bilo koji drugi jezik). Poslednje dugme će zatvarati dijalog.

Primer kontrole Button

Vežba br. 21.

Napravite novi projekat konzolne aplikacije birajući File | New | Project... iz menija:

1. Izaberite Visual C# Projects direktorijum unutar prozora Project Types:, i tip projekta Windows Application u okviru prozora Templates: U okviru za tekst Location: promenite putanju u **C:\Temp\SoftIng\LecturesCode\Vezba21** (ovaj će direktorijum biti automatski napravljen ukoliko već ne postoji), i ostavite podrazumevani tekst u okviru za tekst Name: **Kontrola Button**.
2. Otvorite paletu Properties i za kontrolu *forme* podesite sledeće osobine:

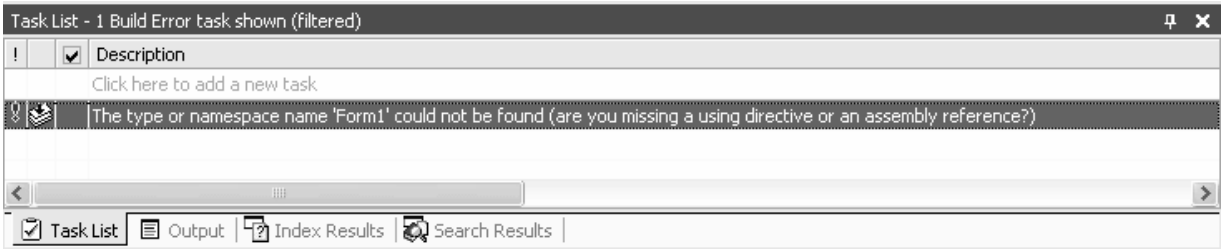
Osobina (Properties)	Vrednost (Value)
Name	fclsMain
Text	Kontrola Button
StartPosition	CenterScreen
Font	Microsoft Sans Serif, Regular, 8, Central European
FormBorderStyle	FixedSingle
Size	262; 150

Napomena:

Vodite računa o konvenciji imenovanja kontrola, kao što je prikazano u lekciji *Imenovanje kontrola, promenljivih i konvencijama pisanja programa.*

3. Probajte odmah posle da startujete aplikaciju, pritiskajući na tastaturi taster F5. Dobićete poruku o greški. Slično ovoj.

4. Takođe u okviru prozora **Task** List generisaće se poruka o grešci. Kliknite dva puta na datu grešku i Visual Studio .NET će nas odvesti do greške u kodu.




```
71      static void Main()
72      {
73          Application.Run(new Form1());
74      }
```

5. Pošto smo promenuli ime kontrole forme, moraćemo ručno promenuti i naziv fome u u samom kodu. Dakle umesto dotičnog naziva Form1 postavite:

```
71      static void Main()
72      {
73        Application.Run(new fclsMain());
74      }
```

Napomena:

Ovo preimenovanje se mora svaki put uraditi samo kod kontrole forme jer Visual Studio .NET sam to ne radi, kao što to radi Programski jezik Delphi. Kod ostalih kontrole naravno to nije potrebno sam VS to radi umesto nas.

6. Pokrenimo ponovo aplikaciju sa F5.
7. Otvorite okvir sa alatima (Toolbar) i tri puta dvostruko pritisnite kontrolu Button. Postavićemo tri kontrole dugmeta na samu formu – Button1, Button2, Button3. Podesite sledeće osobine redom za kontrole dugmad.

za Button1

Osobina (Properties)	Vrednost (Value)
Name	btnEngleski
Text	&Engleski
Cursor	Hand
Image	C:\Program Files\Microsoft Visual Sudio .NET\Common7\Graphics\icons\Flags\ flguk.ico
ImageAlign	BottomLeft
Location	16; 24
Size	96; 23

za Button2

Osobina (Properties)	Vrednost (Value)
Name	btnDanski
Text	&Danski
Cursor	Hand
Image	C:\Program Files\Microsoft Visual Sudio .NET\Common7\Graphics\icons\Flags\ flgden.ico
ImageAlign	BottomLeft
Location	144; 24
Size	96; 23

za Button3

Osobina (Properties)	Vrednost (Value)
Name	btnOk
Text	&OK
Cursor	Hand
Location	88; 72
Size	75; 23

Trebalo bi da dobijemo izgled forme kao na slici:

8. Sada moramo da unesemo šta će aplikacija da radi. Odnosno treba da unesemo kod. Za kontrolu **btnEngleski**. To se može uraditi na dva načina. Prvi tako što dva puta kliknete na dugme **btnEngleski** i VS će sam napraviti događaj **Click**. Drugi način je preko okvira prozora **Events** i dvaput kliktanjem na događaj Click. Unesite sledeći kod u telu Click događaja.

```
123     private void btnEngleski_Click(object sender, System.EventArgs e)
124     {
125       this.Text = "Do you speak English?";
126     }
```

To isto uradite i sa btnDanski.

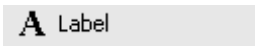
```
128     private void btnDanski_Click(object sender, System.EventArgs e)
129     {
130       this.Text = "Taler du dansk?";
131     }
```

Na kraju to isto uradite i za dugme btnOk.

```
133     private void btnOK_Click(object sender, System.EventArgs e)
134     {
135       Application.Exit();
136     }
```

9. Na kraju pokrenite sa F5 aplikaciju i isprobajte je.

Kontrole Label i LinkLabel



Kontrola Label se možda najčešće koristi od svih kontrola. Pogledajte bilo koju Windows aplikaciju, i naći ćete ovu kontrolu u skoro svakom dijalogu.

Visual Studio .NET sadrži dve kontrole za oznake, koje se predstavljaju korisnicima na dva različita načina;

- Label, standardna Windows oznaka;
- LinkLabel, oznaka slična standardnoj (i izvedena iz nje), ali predstavlja sebe kao internet vezu (hiperveza).

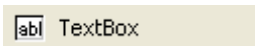
Obe kontrole nalaze se na vrhu kartice Windows Forms. Na slici koja sledi ubačene su obe oznake, da biste videli razliku medu njima.

Ako imate iskustva sa Visual Basicom, primećujete da se za prikazivanje teksta koristi svojstvo Text, a ne svojstvo Caption. Videćete da sve .NET kontrole koriste svojstvo Text za unos teksta koji opisuje kontrolu. Pre .NET-a, svojstva Caption i Text su se upotrebljavala u iste svrhe.

Ovo je otprilike sve što se tiče korišćenja kontrola Label. Obično ne morate dodavati nikakav upravljački kod za događaje, kada je u pitanju standardna labela. U slučaju kontrole LinkLabel, potreban je dodatni kod ako želite da omogućite korisniku da pritisne na ovu oznaku, što će mu omogućiti otvaranje Web stranice.
Kontrola Label ima iznenađujući broj svojstava koje se mogu podešavati. Većina njih je izvedena iz klase Control, ali postoje i neke nove. Tabela koja sledi opisuje najčešće korišćene. Ako drugačije nije naznačeno, svojstvo važi i za kontrolu Label i za LinkLabel:

Ime osobine	Dostupnost	Opis
BorderStyle	Čitanje/upis	Omogućava da izaberete date ivice oznake. Podrazumeva se da nema ivica.
DisabledLinkColor	Čitanje/upis	Samo za LinkLabel: boja koju će kontrola LinkLabel imati kada je korisnik pritisne.
FlatStyle	Čitanje/upis	Definiše izgled kontrole. Ako podesite ovu osobinu na PopUp, kontrola će biti ravna sve dok pokazivač miša ne pomerite na kontrolu. Tada će se kontrola izdići.
Image	Čitanje/upis	Omogućava biranje slike (bitmape, ikone itd.) koja će se pojaviti na oznaci.
ImageAlign	Čitanje/upis	Određuje položaj slike na oznaci.
LinkArea	Čitanje/upis	Samo za LinkLabel: opseg teksta koji će biti prikazan kao veza.
LinkColor	Čitanje/upis	Samo za LinkLabel: Boja hiperveze.
Links	Čitanje/upis	Samo za LinkLabel: kontrola LinkLabel može da drži vise od jedne veze. Ovo svojstvo omogućava vam da pronađete vezu koju hoćete. Kontrola vodi računa o vezama koje se nalaze u tekstu.
LinkVisited	Čitanje/upis	Samo za LinkLabel: obaveštava da li je neka hiperveza prethodno posećena ili ne.
Text	Čitanje/upis	Tekst koji se pojavljuje u oznaci.
TextAlign	Čitanje/upis	Određuje položaj teksta na kontroli.

Kontrola TextBox



Okvir za tekst trebalo bi da bude korišćen kada očekujete od korisnika unos teksta, o kojem ne znate ništa u toku dizajniranja programa (npr. ime korisnika). Osnovna funkcija okvira za tekst jeste da omogući korisniku da unese tekst, pri čemu se može uneti bilo koji znak i sasvim je moguće primorati korisnika da unese samo numeričke vrednosti.
U .NET okruženju postoje dve osnovne kontrole koje mogu primati tekst kao ulazni podatak: TextBox i RichTextBox (o kontroli RichTextBox pričaćemo kasnije u ovom predavanju). Obe kontrole izvedene su iz klase TextBoxBase, koja je izvedena iz klase Control.

Klasa TextBoxBase omogućava osnovnu funkcionalnost za obradu teksta u okvirima za tekst, kao što su izbor, isecanje i kopiranje teksta, kao i veliki broj događaja. Nećemo se sada mnogo baviti posledicama, već demo pogledati jednostavniju od dve kontrole - TextBox. Uradićemo primer da demonstriramo osobine kontrole TextBox, a kasnije dopuniti taj primer i pokazati kontrolu RichTextBox.

Svojstva kontrole TextBox

Kao što već znate, postoji jako mnogo svojstava. Ponovo demo navesti samo one najvažnije:

Ime osobine	Dostupnost	Opis
Causes Validation	Čitanje/upis	Određuje da li će kontrola TextBox menjati slova u mala ili velika. Moguće vrednosti su: Lower: sva uneta slova biće konvertovana u mala slova.
CharacterCasing	Čitanje/upis	Q Normal: neće biti nikakvih promena.
MaxLength	Čitanje/upis	□ Upper: sva uneta slova bide konvertovana u velika slova. Vrednost koja određuje maksimalan broj karaktera u tekstu koji je unet u TextBox. Podesite ovu vrednost na nulu ako je maksimum ograničen samo raspoloživom memorijom.
MultiLine	Čitanje/upis	Pokazuje da li je ovo kontrola sa vise redova. Kontrola sa vise redova može da prikaže vise redova teksta.
PasswordChar	Čitanje/upis	Određuje da li karakter lozinke treba da zameni stvarne karaktere koji su uneti u okvir za tekst sa jednim redom. Ako je svojstvo Multiline postavljeno na true, tada ovo svojstvo nema nikakvog efekta.
Readonly	Čitanje/upis	Logička vrednost koja pokazuje da li je tekst samo za čitanje.
ScrollBars	Čitanje/upis	Određuje da li tekst u vise redova treba da ima traku za pomeranje.
SelectedText	Čitanje/upis	Tekst koji je izabran u okviru za tekst.
SelectionLength	Čitanje/upis	Broj izabranih karaktera u tekstu. Ako je ova vrednost veća od ukupnog broja karaktera u tekstu, onda kontrola resetuje ovu vrednost na ukupan broj karaktera, minus vrednost koja se nalazi u svojstvu SelectionStart.
SelectionStart		Početak izabranog teksta u okviru za tekst.
Wordwrap		Određuje da li u okvirima za tekst sa vise redova, reč koja premašuje širinu kontrole treba automatski da bude prebačena u novi red.

Događaji kontrole TextBox

Pažljiva provera ispravnosti teksta, koji se unosi u kontrolu TextBox, može da napravi razliku između zadovoljnih i veoma ljutih korisnika.

Verovatno ste iskusili koliko iritirajuće može biti kada dijalog proverava ispravnost svog sadržaja tek kada pritisnete OK. Ovaj način provere ispravnosti podataka obično za posledicu ima pojavljivanje poruke da su podaci u „kontroli TextBox broj tri" neispravni. Tada možete nastaviti da pritiskate OK sve dok svi podaci ne postanu korektni. Očigledno, ovo nije dobar način provere ispravnosti podataka. Dakle, šta se može učiniti po ovom pitanju?

Odgovor leži u obradi događaja za proveru ispravnosti, koji obezbeđuje kontrola TextBox. Ako želite da budete sigurni da se ne unose nedozvoljeni karakteri ili da su dopušteni samo karakteri iz određenog opsega, onda ćete hteti da korisnik kontrole bude obavešten da li je uneti karakter ispravan ili nije.

Kontrola TextBox obezbeđuje ove događaje (koji su svi izvedeni iz klase Control):

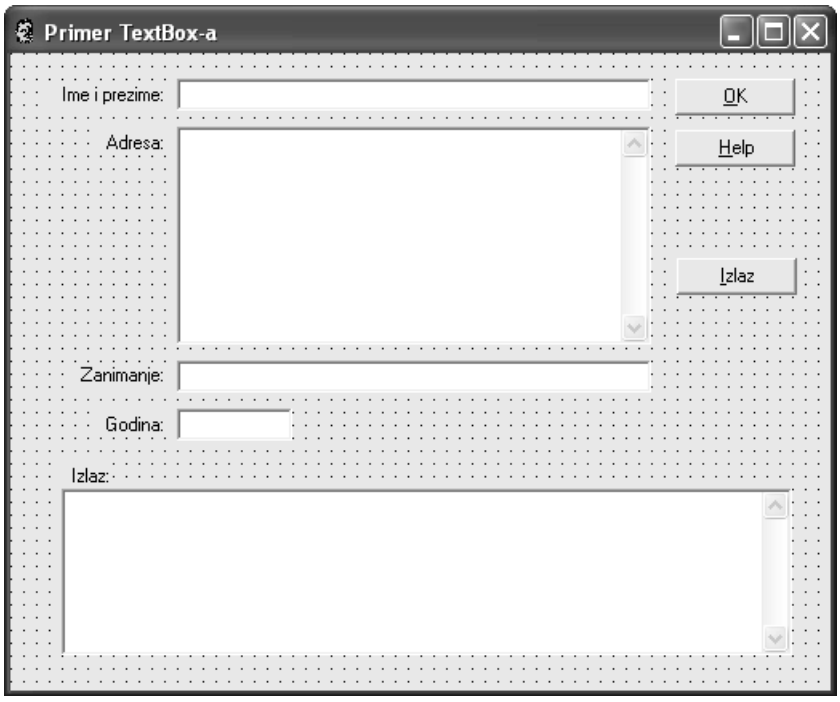
Ime događaja	Opis
Enter	Ovih šest događaja dešavaju se redom kojim su ovde navedeni. Poznati su kao „događaji fokusa" i aktiviraju se uvek kada se menja fokus kontrole, uz dva izuzetka. Događaji Validating i Validated dešavaju se samo kada kontrola koja prima fokus, ima svojstvo CausesValidation podešeno na true. Razlog da se događaj veže za kontrolu koja prima fokus, jeste taj što nekad ne želite proveru ispravnosti sadržaja kontrole, čak i kada se fokus menja. Primer za ovo je kada korisnik pritisne Help dugme.
GotFocus	
Leave	
Validating	
Validated	
LostFocus	
KeyDown	Sledeća tri događaja su poznatija kao „taster događaj". Oni omogućavaju da pratite i menjate ono što je uneto u kontrolu. KeyDown i KeyUp primaju kod tastera koji je pritisnut. Na ovaj način možete videti da li su pritisnuti specijalni tasteri, kao sto su *Shift*, *Control* ili **F1**. KeyPress, sa druge strane, prima karakter koji odgovara tastera. To znači da vrednost za slovo „a" nije ista kao i za slovo Ovo može biti korisno ako želite da ograničite opseg karaktera, npr. omogućavanjem unosa samo numeričkih karaktera.
KeyPress	
KeyUp	
Change	Dešava se uvek kada se tekst u okviru za tekst menja, bez obzira na vrstu promene.

Primer kontrole TextBox

Vežba br. 22.

Kreiraćemo dijalog u okviru koga možete da unesete svoje ime, adresu, zanimanje i godine starosti. Svrha ovog primera jeste da vas pripremi za korišćenje svojstava i događaja, a ne da kreira nešto veoma korisno.
Prvo ćemo napraviti korisnički interfejs:

1. Izaberite Visual C# Projects direktorijum unutar prozora Project Types:, i tip projekta Windows Application u okviru prozora Templates: U okviru za tekst Location: promenite putanju u **C:\Temp\SoftIng\LecturesCode\Vezba22** (ovaj će direktorijum biti automatski napravljen ukoliko već ne postoji), i ostavite podrazumevani tekst u okviru za tekst Name: **PrimerTextBox.**
2. Napravite formular, kao na slici koja sledi, dodavanjem oznaka, okvira za tekst i dugmeta. Pre nego što promenite veličinu okvira za tekst - txtAddress i txtOutput ~ kao sto je pokazano, morate da podesite njihova svojstva Multiline na true. Ovo ćete uraditi desnim pritiskom na kontrole i biranjem Properties:



3. Kontrole nazovite kao što je prikazano u tabelama.

za Form1

Osobina (Properties)	Vrednost (Value)
Name	fclsMain
Text	Primer TextBox-a
Font	Microsoft Sans Serif, Regular, 8, Central European
FormBorderStyle	FixedSingle
StartPosition	CenterScreen
Size	520; 432

pet kontrola TextBox, redom:

za TextBox1

Osobina (Properties)	Vrednost (Value)
Name	txtIme
Text	
Multiline	False
Location	104; 16
Size	296; 20

za TextBox2

Osobina (Properties)	Vrednost (Value)
Name	txtAdresa
Text	
Multiline	True
ScrollBars	Vertical
Location	104; 46
Size	296; 136

za TextBox3

Osobina (Properties)	Vrednost (Value)
Name	txtZanimanje
Text	
Multiline	False
Location	104; 192
Size	296; 20

za TextBox4

Osobina (Properties)	Vrednost (Value)
Name	txtGodiste
Text	
Multiline	False
Location	104; 222
Size	72; 20

za TextBox5

Osobina (Properties)	Vrednost (Value)
Name	txtIzlaz
Text	
Multiline	True
ReadOnly	True
ScrollBars	Vertical
Location	0; 296
Size	512; 104

Pored njih 5 kontrola Label-a, kao što je prikazano na prethodnoj slici.

Zatim na formu postavite tri dugmeta odnosno kontrole Button, redom:

za Button1

Osobina (Properties)	Vrednost (Value)
Name	btnOK
Text	&OK
Cursor	Hand
Location	415; 16
Size	75; 23

za Button2

Osobina (Properties)	Vrednost (Value)
Name	btnHelp
Text	&Help
Cursor	Hand
CausesValidation	false
Location	415; 48
Size	75; 23

za Button3

Osobina (Properties)	Vrednost (Value)
Name	btnIzlaz
Text	&Izlaz
Cursor	Hand
Location	416; 128
Size	75; 23

Setite se iz razmatranja događaja Validating i Validate da podešavanje ove osobine na false dopušta korisniku da pritisne na dugme, a da ne mora da brine o ispravnosti podataka.

Kad smo podesili dimenzije forme, i na njoj kontrole, podesimo svojstvo Anchor kontrola tako da se kontrole ponašaju normalno kada menjamo veličinu formulara (naravno, ako vratimo osobinu forme FormBorderStyle na Sizable). Podesite svojstvo Anchor kao što je prikazano u tabeli koja sledi:

Kontrola	Anchor vrednost
Sve oznake – labele	Top, Left
Svi okviri za tekst- TextBox-ove osim txtAdresa	Top, Bottom, Left, Right Top, Left, Right
Tri dugmeta - Button	Top, Right

Savet: Tekst koji opisuje svojstvo Anchor možete kopirati iz jedne kontrole u drugu; na taj način kopirate i vrednosti osobine. Ovako ne morate koristiti padajući meni.

Razlog zbog kojeg je kontrola **txtOutput** stavljena na dno formulara (Anchor), a ne pripojena uz ivicu (Dock), jeste taj što želimo da ovaj prostor za prikaz unetog teksta menja svoju veličinu zajedno sa promenom veličine formulara. Da smo kontrolu pripojili donjoj ivici formulara pri

promeni veličine formulara, kontrola bi uvek ostajala pripojena ivici, ali njena veličina ne bi bila promenjena.

Ako smo vratili osobinu forme FormBorderStyle na Sizable onda bi trebalo da podesimo osobine Size i MinSize. Ne bi imalo smisla da formular bude manji nego što je sada. Stoga, podesite da svojstvo MinSize bude isto kao svojstvo Size.

Podešavanje izgleda formulara ovim je završeno. Ako pokrenete ovaj program, ništa se ne dešava kada pritisnete na neko dugme ili ukucate neki tekst. Ali, ako povećate veličinu formulara ili je povučete, primetićete da se kontrole ponašaju baš onako kako smo hteli - ostaju u željenom položaju i odgovarajućoj razmeri. Ako ste ikada pokušali da obavite isti zadatak u jeziku kao što je Visual Basic 6, znate koliki ste trud upravo uštedeli.
Sada je vreme da predemo na kod. Pritisnite desnim tasterom miša formular i izaberite View Code. Ako imate prikazan okvir sa alatima, trebalo bi da ga sklonite da biste dobili vise mesta za prozor sa kodom.

Iznenađujuće malo koda se pojavljuje u programu za uređivanje. Na vrhu naše klase su definisane kontrole, ali tek kada proširite oblast pod oznakom Windows Form Designer Generated Code, možete da vidite šta ste sve do sada uradili. Ovaj kod nikada ne bi trebalo da menjate! Sledeći put kada nešto promenite u dizajneru, novi k6d će biti upisan preko postojećeg, ili, još gore, možete napraviti takvu izmenu da dizajner vise ne pokazuje vaš formular.

Posvetite malo vremena iskazima koji se nalaze u ovoj oblasti. Shvatićete zašto je moguće kreirati Windows aplikaciju bez korišćenja Visual Studio .NET-a. Sve što se ovde nalazi jednostavno je moglo biti kreirano i u Notepadu ili nekom drugom programu za uređivanje teksta.

Takođe, videćete zašto se ovo ne preporučuje. Voditi računa o svemu što vidite je teško; lako se potkradaju greške i, zbog toga što ne možete da vidite efekte onoga što radite, smeštanje kontrola na formular na pravi način predstavlja naporan posao. Ovo, međutim, daje šansu nekim drugim proizvođačima softvera za pisanje sopstvenog programerskog okruženja konkurentskog Visual Studio .NET-u, zbog toga što su kompajleri, koji se koriste za kreiranje formulara, sastavni deo .NET okruženja, a ne Visual Studio .NET-a.

Dodavanje rutina za obradu događaja

Možemo još malo iskoristiti dizajner formulara, pre nego što dodamo naš kod. Vratite se dizajner formulara (pritiskom na jezičak kartice u vrhu programa za uređivanje teksta), i dvaput pritisnite dugme btnOK. Ponovite ovo i sa drugim dugmetom. Kao što smo već videli iz primera za dugme, dvostruki pritisak na dugme kreira rutinu za obradu događaja Click. Kada se pritisne na OK dugme, želimo da prebacimo tekst iz ulaznih okvira za tekst u izlazni okvir koji je samo za čitanje.
Ovo je kod za dva događaja Click:

```
private void btnOK_Click(object sender, System.EventArgs e)
{
  // Nema provere ispravnosti, pošto nije ni potrebna
  string izlaz;
  //Spajamo tekstove cetiri textbox-a
  izlaz = "Ime i prezime:" + this.txtIme.Text + "\r\n";
  izlaz += "Adresa:" + this.txtAdresa.Text + "\r\n";
  izlaz += "Zanimanje:" + this.txtZanimanje.Text + "\r\n";
  izlaz += "Godina:" + this.txtGodiste.Text;
  // Ubacujemo novi tekst
  this.txtIzlaz.Text = izlaz;
}

private void btnHelp_Click(object sender, System.EventArgs e)
{
  // Pisemo kratak opis svakog textbox-a u izlaznom textbox-u
  string izlaz;
  izlaz = "Ime i prezime = Tvoje ime i prezime\r\n";
  izlaz += "Adresa = Tvoja adresa\r\n";
  izlaz += "Zanimanje = Jedino dozvoljen unos je: Programer\r\n";
  izlaz += "Godina = Godina rodenja";
  //Ubacimo novi tekst
  this.txtIzlaz.Text = izlaz;
}
```

U obe funkcije koristi se svojstvo Text okvira za tekst, bilo da se čita ili upisuje u funkciji btnOK_Click (), ili da se samo upisuje u akciji btnHelp_Click () .
Ubacujemo informacije koje je korisnik uneo, bez provere njihove korektnosti. To znači da proveru moramo raditi na nekom drugom mestu. U ovom primeru postoje određeni kriterijumi koji se moraju ispuniti, da bi podaci bili korektni:

- Polje za ime korisnika ne može biti prazno;
- Broj godina korisnika mora biti veći ili jednak nuli;
- Zanimanje korisnika mora biti „Programer" ili ostavljeno prazno;
- Polje za adresu korisnika ne može biti prazno.

Ovde vidimo da su provere za dva okvira za tekst (txtName i txtAddress) iste. Takođe, moramo sprečiti korisnika da unese nevažeći podatak u polju Age, i na kraju moramo proveriti da li je korisnik programer.
Da bismo sprečili korisnika da pritisne dugme OK, nego što je bilo šta uneto, podesićemo svojstvo Enabled dugmeta OK na false u konstruktoru formulara, vodeći računa da u ovo svojstvo ne upišemo vrednost true sve dok se ne pozove kod u metodu Initialize-Component().

```
public fclsMain()
{
  //
  // Required for Windows Form Designer support
  //
  InitializeComponent();
  this.btnOK.Enabled = false;
```

Sada ćemo kreirati rutinu za obradu događaja za dva okvira za tekst, za koje moramo proveriti da li su prazni. Ovo radimo korišćenjem događaja Validating za okvire. Kontrolu obaveštavamo da će događaj biti obrađen u metodi pod imenom txtBoxEmpty_Validating ().
Takođe, trebalo bi da znamo status kontrola. Za ovo koristimo svojstvo **Tag** okvira za tekst. Ako se sećate, rekli smo da se, iz dizajnera formulara, svojstvu Tag mogu dodeliti samo stringovi. Međutim, pošto podešavamo svojstvo Tag iz koda, možemo uraditi bilo šta sa njim. Ovde je bolje koristiti logičku vrednost.
U konstruktor dodajemo sledeće iskaze:

```
36        InitializeComponent();
37        this.btnOK.Enabled = false;
38
39        // Podesavamo vrednost Tag na false zbog testiranja
40        // podataka da li su ispravno uneta.
41        this.txtAdresa.Tag = false;
42        this.txtGodiste.Tag = false;
43        this.txtIme.Tag = false;
44        this.txtZanimanje.Tag = false;
45
46        // Prijavljujemo dogadjaje
47        this.txtIme.Validating += new
48          System.ComponentModel.CancelEventHandler(this.txtBoxPrazan_Validacija);
49        this.txtAdresa.Validating += new
50          System.ComponentModel.CancelEventHandler(this.txtBoxPrazan_Validacija);
```

Za razliku od rutina za obradu događaja dugmeta, koje smo već videli, rutina za obradu događaja Validating je specijalna verzija standardne ratine za obradu događaja System.Event-Handler. Specijalna rutina za obradu događaja Validating treba nam zbog slučaja neispravnih podataka, kada treba sprečiti dalju obradu tih podataka. Kada bismo obustavili dalju obradu svih podataka, to bi praktično značilo da bi bilo nemoguće napustiti okvir za tekst sve dok se ne unesu ispravni podaci. Nećemo raditi ništa tako drastično u ovom primeru.

Događaji Validating i Validate, zajedno sa svojstvom CausesValidation rešavaju problem koji nastaje korišćenjem događaja GotFocus i LostFocus za proveru ispravnosti sadržaja kontrole. Problem nastaje kada se događaji GotFocus i LostFocus neprestano aktiviraju, zbog toga sto kod za proveru ispravnosti prebacuje fokus sa jedne na drugu kontrolu. To je stvaralo beskonačnu petlju.

Dodajemo rutinu za obradu događaja:

```
317
318       private void txtBoxPrazan_Validacija(object sender,
319         System.ComponentModel.CancelEventArgs e)
320       {
321         // Znamo da je posiljalac kontrola TextBox,
322         // pa konvertujemo objekat sender u taj tip
323         TextBox tb;
324         tb = (TextBox)sender;
325         // Ako je tekst nije unet setujemo boju pozadine u crvenc
326         // To radimo koristeci Tag vrednost kontrola pomocu koje
327         // proveravamo validnost.
328         if (tb.Text.Length == 0)
329         {
330           tb.BackColor = Color.Red;
331           tb.Tag = false;
332           // U slucaju da hocemo da nastavimo sa daljom obradom
333           // kao sto necemo daodali bismo sledecu liniju
334           // e.Cancel = true;
335         }
336         else
337         {
338           tb.BackColor = System.Drawing.SystemColors.Window;
339           tb.Tag = true;
340         }
341
342         // Na kraju zovemo rutinu ValidacijaZasSve koja setuje
343         // osobinu Enabled za OK dugme.
344         ValidacijaZaSve();
345       }
346
```

Zato što vise okvira za tekst koriste ovaj metod za obradu događaja, ne možemo biti sigurni koji okvir poziva funkciju. Ipak, znamo da efekat izvršavanja ove funkcije treba uvek da bude isti, bez obzira na to koji okvir poziva funkciju. Stoga, konvertujemo parametar sender u objekat tipa TextBox i dalje nastavljamo da radimo na tom objektu.
Ako je dužina teksta u okviru nula, podesimo boju pozadine na crveno i tag na false. Ako ne, onda podesimo boju pozadine na standardnu Windows boju prozora.

Uvek koristiti boje iz enumeracije System.Drawing.SystemColors kada želite da podesite kontrolu na standardnu boju. Ako podesite boju na belu, vaša aplikacija će izgledati čudno u odnosu na ostale aplikacije ako je korisnik promenio podrazumevane boje.

Funkcija ValidateAll () je opisana na kraju ovog primera

Nastavljamo sa događajem Validating dodajući rutinu za obradu kontroli txtZanimanje. Procedura je ista kao za dve prethodne rutine, ali je kod drugačiji zato što zanimanje mora biti ili Programer, ili ostavljeno prazno, da bi kontrola bila validna. Stoga, dodajemo novi red konstruktoru:

```
51          this.txtZanimanje.Validating += new
52            System.ComponentModel.CancelEventHandler(this.txtZanimanje_Validacija);
```

Zatim, i rutinu za obradu:

```
356
357       private void txtZanimanje_Validacija(object sender,
358         System.ComponentModel.CancelEventArgs e)
359       {
360         // Konvertujemo objekat sender u tip TextBox
361         TextBox tb = (TextBox)sender;
362
363         // Proveravamo da li su  vrednosti korektne
364         if (tb.Text.CompareTo("Programer") == 0 || tb.Text.Length == 0)
365         {
366           tb.Tag = true;
367           tb.BackColor = System.Drawing.SystemColors.Window;
368         }
369         else
370         {
371           tb.Tag = false;
372           tb.BackColor = Color.Red;
373         }
374
375         // Podesavamo stanje dugmeta OK
376         ValidacijaZaSve();
377       }
```

Pretposlednji izazov je polje za godine starosti. Ne želimo da korisnik unosi ništa drugo osim pozitivnih brojeva (uključujući i nulu, da bi primer bio jednostavniji). Da bismo ovo postigli, koristićemo događaj KeyPress za uklanjanje neželjenih znakova pre nego što se oni prikažu u okvira za tekst.

Prvo se prijavljujemo za događaj KeyPress, na sličan način kao i sa prethodnim rutinama - u konstruktoru:

```
53        this.txtGodiste.KeyPress += new
54          System.Windows.Forms.KeyPressEventHandler(this.txtGodiste_KeyPress);
```

Rutina za obradu je takođe specijalna. Obezbeđen je parametar System.Windows.Forms. KeyPressEventHandler, zato što je događaju potrebna informacija o tome koji je taster pritisnut. Dodajemo rutinu za obradu događaja:

```
379       private void txtGodiste_KeyPress(object sender,
380         System.Windows.Forms.KeyPressEventArgs e)
381       {
382         if ((e.KeyChar < 48 || e.KeyChar > 57) && e.KeyChar != 8)
383           e.Handled = true; // Uklanja znak
384       }
```

ASCII vrednosti za cifre od 0 do 9 su u opsegu od 48 do 57. Ipak, pravimo jedan izuzetak. ASCII vrednost 8 odnosi se na taster za brisanje *(Backspace)*. Neophodno je dozvoliti i pritiskanje tastera za brisanje ako želimo da dopustimo korisniku da ispravi netačnu informaciju.

Podešavanjem svojstva Handled objekta KeyPressEventArgs na true, govorimo kontroli da ne radi ništa sa znakom; zbog toga znak neće bid prikazan.

Za sada, kontrola nije označena ni kao ispravna, ni kao neispravna. To je zato što nam treba još jedna provera da vidimo da li je išta uneto. S obzirom na to da već imamo metodu za ovu proveru, jednostavno prijavljujemo događaj Validating i za kontrolu Age, dodavanjem još jedne linije u konstruktor:

```
this.txtGodiste.Validating += new
  System.ComponentModel.CancelEventHandler(this.txtBoxPrazan_Validacija);
```

Ostaje još jedna stvar o kojoj moramo da proverimo. Ako korisnik unese ispravne informacije u sve okvire za tekst, a zatim napravi izmenu tako da neka informacija postane neispravna, dugme OK ostaje dostupno. Dakle, moramo napisati poslednju rutinu za obradu događaja za sve tekstualne okvire: događaj Change koji će dugme OK dugme učiniti nedostupnim ako bilo koje polje za tekst sadrži neispravne podatke.
Događaj Change aktivira se kad god se tekst u kontroli menja. Prijavljujemo događaj dodavanjem sledećih redova konstruktoru:

```
this.txtIme.TextChanged += new
  System.EventHandler(this.txtBox_TextPromenjen);
this.txtAdresa.TextChanged += new
  System.EventHandler(this.txtBox_TextPromenjen);
this.txtGodiste.TextChanged += new
  System.EventHandler(this.txtBox_TextPromenjen);
this.txtZanimanje.TextChanged +=
  new System.EventHandler(this.txtBox_TextPromenjen);
```

Događaj Change koristi standardnu rutinu za obradu događaja, koja nam je poznata iz događaja Click. Zatim dodajemo i sam događaj:

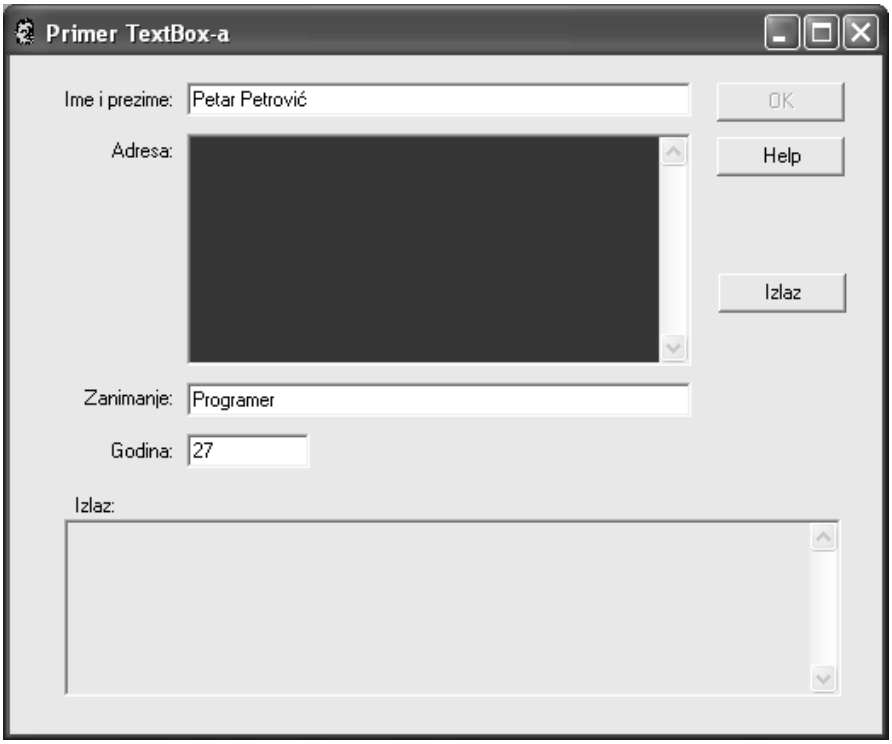
```
386     private void txtBox_TextPromenjen(object sender, System.EventArgs e)
387     {
388       // Konvertujemo parametar sender u tip TextBox
389       TextBox tb = (TextBox)sender;
390
391       // Proveravamo ispravnost podataka i
392       // podesavamo boju boju pozadine.
393       if (tb.Text.Length == 0 && tb != txtZanimanje)
394       {
395         tb.Tag = false;
396         tb.BackColor = Color.Red;
397       }
398       else if (tb == txtZanimanje &&
399         (tb.Text.Length != 0 && tb.Text.CompareTo("Programer") != 0))
400       {
401         // Ovde se ne podesava boja, jer ce se menjati kako
402         // korisnik upisue slova
403         tb.Tag = false;
404       }
405       else
406       {
407         tb.Tag = true;
408         tb.BackColor = SystemColors.Window;
409       }
410
411       // Podesavamo dugme OK
412       ValidacijaZaSve();
413     }
```

Ovoga puta moramo tačno znati koja kontrola poziva ratinu za obradu događaja, zato što ne želimo da se boja pozadine okvira Occupation promeni u crveno kada korisnik započne kucanje.

Ovo postižemo proverom svojstva Name onog okvira koji je prosleđen kroz parametar sender.
Još jedna stvar ostaje: metoda ValidateAll() koja dugme OK čini dostupnim ili nedostupnim:

```
347     private void ValidacijaZaSve()
348     {
349       // Setovanje osobine Enabled dugmeta OK
350       // na true ako su svi Tag vrednosti true.
351       this.btnOK.Enabled = ((bool)(this.txtAdresa.Tag) &&
352         (bool)(this.txtGodiste.Tag) &&
353         (bool)(this.txtIme.Tag) &&
354         (bool)(this.txtZanimanje.Tag));
355     }
```

Ova metoda jednostavno podešava svojstvo Enabled za dugme OK na true, ako sva svojstva **Tag** imaju vrednost true. Moramo vrednosti svojstava **Tag** konvertovati u logičke vrednosti, zato što je svojstvo Enabled logička promenljiva.
Ako isprobate program, trebalo bi da dobijete nešto nalik ovome:



Primetićete da možete da pritisnete na dugme Help dok imate okvire za tekst sa neispravnim podacima, bez promene boje pozadine u crveno.

Primer koji ste upravo završili jeste prilično veliki u odnosu na sledeće primere iz ovog predavanja. To je zato što ćemo ovaj primer koristiti za sve koji slede.

Korišćenje kontrola za Windows forme - II deo

Kontrole RadioButton i CheckBox

Kao što smo već rekli, kontrole RadioButton i CheckBox koriste istu osnovnu klasu kao i kontrola Button, iako im se izgled i korišćenje znamo razlikuju.
Radio dugme se tradicionalno prikazuje kao oznaka sa kružićem na levoj strani, koje može **biti** izabrano ili ne. Radio dugme bi trebalo koristiti kada želite od korisnika da bira između međusobno isključivih opcija. Primer za to bi mogao biti pol korisnika.
Da biste grupisali vise radio dugmadi kako bi ona činila logičku jedinicu, koristite kontrolu GroupBox. Prvo stavite kontrolu GroupBox na formular, zatim ubacite neophodne kontrole RadioButton unutar tog okvira; kontrole RadioButton znaće kako da menjaju svoj status tako da samo jedna kontrola unutar grupnog okvira bude izabrana. Ako kontrole RadioButton ne stavite u grupni okvir, samo jedna od njih unutar formulara može biti izabrana.
Kontrola CheckBox (polje za potvrdu) tradicionalno se predstavlja kao oznaka sa kvadratićem na levoj strani, koji može biti potvrđen. Koristite polja za potvrdu kada želite da omogućite korisniku da bira jednu ili vise opcija. Primer može biti upitnik gde korisnik bira operativne sisteme koje je koristio (npr. Windows 95, Windows 98, Linux, Max OS X itd.).
Pogledaćemo važnije osobine i događaje za ove dve kontrole, počevši od radio dugmeta, a zatim pred na primere.

Svojstva kontrole RadioButton

Pošto je kontrola izvedena is klase ButtonBase, a već smo videli neke osobine kontrole Button, ovde ćemo opisati samo nekoliko svojstava. Kao i uvek, ako vam je potrebna kompletna lista svojstava, pogledajte MSDN biblioteku.

Ime osobine	Dostupnost	Opis
Appearance	Čitanje/upis	Radio dugme se pojavljuje kao oznaka sa kružićem levo, u sredini ili desno od oznake, ili kao standardno dugme. U poslednjem slučaju, dugme se prikazuje kao pritisnuto ako je izabrano, odnosno u 3D obliku ako nije.
AutoCheck	Čitanje/upis	Ako je ovo svojstvo podešena na true, znak za potvrđivanje se pojavljuje kada korisnik pritisne na radio dugme. Kada je ovo svojstvo podešeno na false, podrazumeva se da se znak za potvrđivanje ne pojavljuje.
CheckAlign	Čitanje/upis	Određuje položaj radio dugmeta. Svojstvo može biti podešeno na left, middle ili right (levo, u sredini, desno).
Checked	Čitanje/upis	Pokazuje status kontrole. Ako je kontrola potvrđena, onda je ovo svojstvo podešeno na true, odnosno false u suprotnom.

Događaji kontrole RadioButton

Obično, imaćete samo jedan događaj kada radite sa kontrolama RadioButton. Naravno, postoje i drugi događaji. Razmatraćemo samo dva u ovom predavanju, a jedini razlog što pominjemo dragi događaj jeste taj što postoji jedva primetna razlika koju treba istaći:

Ime	Opis
CheckChanged	Ovaj događaj se šalje kada se menja potvrda radio dugmeta. Ako ima vise kontrola RadioButton na formularu ili unutar jednog grupnog okvira, ovaj događaj se šalje dva puta: prvo kontroli koja je bila potvrđena a sada vise nije, a zatim kontroli koja postaje potvrđena.
Click	Ovaj događaj se šalje svaki put kada se pritisne na radio dugme. Ovo nije isto kao i događaj CheckChanged, zato što pritisak na radio dugme dva ili vise puta uzastopno menja svojstvo Checked samo jednom - i to samo ako kontrola nije prethodno bila potvrđena.

Svojstva kontrole CheckBox

Kao što pretpostavljate, osobine i događaji ove kontrole su slični onima za RadioButton, ali tu su i dve nove:

Ime osobine	Dostupnost	Opis
CheckState	Čitanje/upis	Za razliku od radio dugmeta, polje za potvrdu može imati tri statusa: Checked, Indeterminate i Unchecked (potvrđeno, neodređeno i nepotvrđeno). Ako polje ima status Indeterminate, onda je obično kružić pored oznake siv, tako pokazujući da je trenutna vrednost kružića ili neispravna ili nema nikakvog smisla u trenutnim okolnostima. Primer ovakvog statusa možete videti ako izaberete nekoliko datoteka u Windows Exploreru i pogledate Properties ovih datoteka. Ako se neke od ovih datoteka mogu samo čitati, a druge i menjati i čitati, onda će polje za potvrdu Read-only biti potvrđeno, ali sivo - neodređeno.
ThreeState	Čitanje/upis	Ako je ovo svojstvo podešena na false, korisnik neće modi da izmeni status kontrole na Indeterminate. Možete, međutim, izmeniti status u kodu.

Događaji kontrole CheckBox

Obično, koristićete samo jedan ili dva događaja za ovu kontrolu. Iako događaj CheckChanged postoji i za kontrole RadioButton i CheckBox, javlja se izvesna razlika u efektima ovih događaja:

Ime	Opis
CheckedChanged	Dešava se kada se menja svojstvo Checked kontrole. Kod kontrola sa osobinom ThreeState podešenom na true moguće je pritisnuti na polje za potvrdu bez promene Checked osobine. Do ovoga dolazi kada se status polja menja iz Checked uIndeterminate.
CheckedStateChanged	Dešava se kada se menja svojstvo CheckedState kontrole. Kako su Checked i Unchecked moguće vrednosti za svojstvo CheckedState, ovaj događaj će biti poslat uvek kada se menja svojstvo Checked. Takođe, biće poslat kada se status menja iz CheckeduIndeterminate.

Ovim završavamo opise događaja i svojstava kontrola RadioButton i CheckBox. Pre nego što pređemo na primer, pogledajmo kontrolu GroupBox, koju smo ranije pomenuli.

Kontrola GroupBox

Pre nego što predemo na primer, pogledaćemo kontrolu GroupBox. Ova kontrola se obično koristi zajedno sa kontrolama RadioButton i CheckBox, da bi napravila okvir i zaglavlje za grupu kontrola koje su logički povezane.

Grupni okvir se jednostavno dovuče na formular, a zatim se stavljaju ostale kontrole unutar tog okvira (obrnuto nije moguće - ne možete staviti grupni okvir na već postojeće kontrole). To za efekat ima da roditelj kontrola postaje grupni okvir, a ne formular, što znači da može biti izabrano vise kontrola RadioButton u isto vreme. Međutim, unutar jednog grupnog okvira može biti izabrano samo jedno radio dugme.

Odnos roditelj/dete možda zahteva dodatno objašnjenje. Kada se kontrola stavi na formular, kaže se da je formular roditelj kontrole, odnosno kontrola je dete formulara. Kada stavite grupni okvir na formular, grupni okvir postaje dete formulara. Kako grupni okvir sadrži nove kontrole, on postaje roditelj za te kontrole. Ovo znači da kada pomerite grupni okvir, pomeraju se i sve kontrole koje se nalaze u tom okviru.

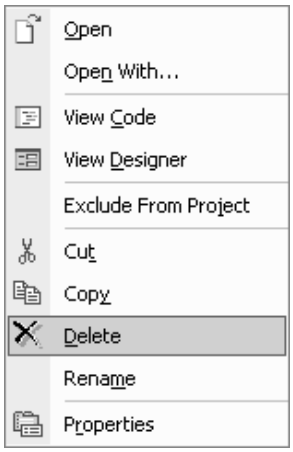
Dragi efekat stavljanja kontrola u grupni okvir jeste mogućnost menjanja određenih svojstava kontrola jednostavnom promenom odgovarajućih svojstava grupnog okvira. Na primer, ako želite da onesposobite sve kontrole unutar grupnog okvira, možete jednostavno podesiti svojstvo Enabled grupnog okvira na false.
Prikazaćemo primenu grupnog okvira u primeru koji sledi.

Primer kontrole RadioButton i CheckButton

Vežba br. 23.

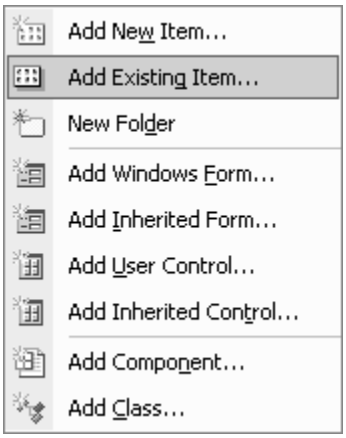
Stari projekat odnosno vežbu pod brojem 22. **Primer TextBox**. Snimite je u drugom folderu, recimo **C:\Temp\SoftIng\LecturesCode\Vezba23.**

Otvorite VS .NET i kreirajte novi projekat. Izaberite Visual C# Projects direktorijum unutar prozora Project Types:, i tip projekta Windows Application u okviru prozora Templates: U okviru za tekst Location: promenite putanju u **C:\Temp\SoftIng\LecturesCode\Vezba23**, i ostavite podrazumevani tekst u okviru za tekst Name: **KontrolaRadioCheck Button.**



Da bi smo iskoristili prethodni primer, prvo ćete obrisati iz već postojećeg projekta fajl **Form1.cs** tako što izaberete pomoću desnog tastera miša opciju **Delete** na selektovan fajl.

Zatim ćemo dodati fajl sa istim nazivom ali iz prethodnog primera odnosno vežbe 19 koje ste malo pre iskopirali.
Selektujte naziv projekta u prozoru Solution Explorera **KontrolaRadioCheck Button,** izaberete pomoću desnog tastera miša opciju **Add>Add Existing Item...** na selektovan naziv projekta. I birate naziv fajla Form1.cs iz foldera **C:\Temp\SoftIng\LecturesCode\Vezba23\ Primer TextBox.**



I ako ste uradili kako treba, možete otvoriti formu iz prethodnog primera.

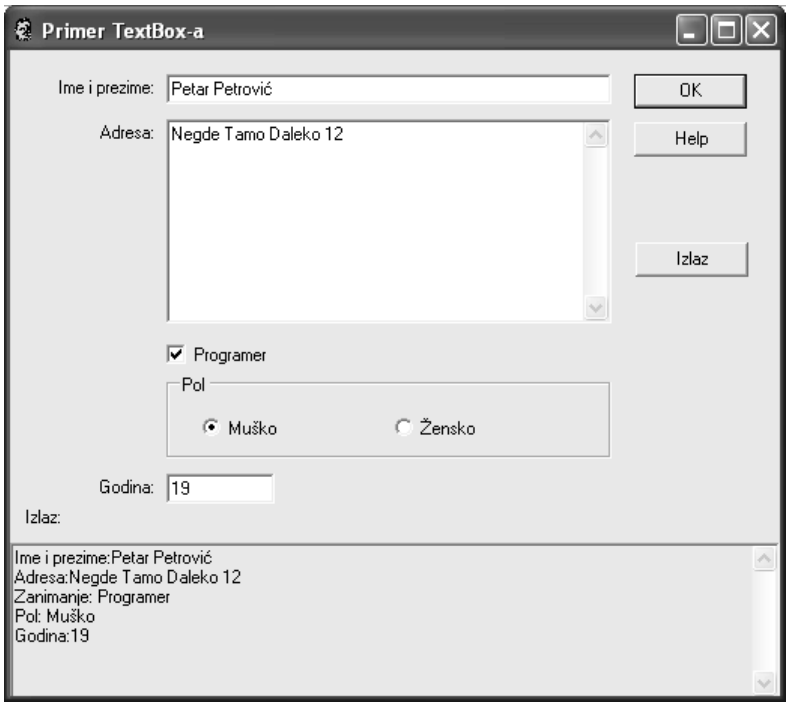
Izmenićemo prethodni primer u kome smo prikazali upotrebu okvira za tekst. U tom primeru jedino moguće zanimanje je bilo Programer. Umesto da primoramo korisnika da ispisuje ovu reč, promenićemo okvir za tekst u polje za potvrdu.

Da bismo demonstrirali korišćenje radio dugmeta, tražićemo od korisnika još jednu informaciju: kojeg je **pola**.

Izmenite okvir sa tekstom na sledeći način:

1. Uklonite kontrolu lblZanimanje i okvir za tekst TextBox pod nazivom txtZanimanje.

2. Promenite veličinu formulara tako da u njega može da stane grupni okvir sa informacijom o polu korisnika, i imenujte nove kontrole kao što je prikazano na slici:



3. Dodajte na formu kontrolu CheckedBox i promenite sledeća svojstva:

za CheckBox1

Osobina (Properties)	Vrednost (Value)
Name	chkProgramer
Text	Programer
Checked	True
Location	104; 191
Size	104; 24

4. Dodajte na formu kontrolu GroupBox i promenite sledeća svojstva:

za GroupBox1

Osobina (Properties)	Vrednost (Value)
Name	grpPol
Text	Pol
Location	104; 216
Size	296; 56

5. Dodajte na formu u GroupBox-u dve kontrole RadioButton i promenite sledeća svojstva:

za RadioButton1

Osobina (Properties)	Vrednost (Value)
Name	rdoMusko
Text	Muško
Checked	True
Location	24; 24
Size	104; 24

za RadioButton2

Osobina (Properties)	Vrednost (Value)
Name	rdoZensko
Text	Žensko
Checked	False
Location	152; 24
Size	104; 24

Primetite da ne mogu obe biti podešene na true. Ako to probate, vrednost osobine drugog radio dugmeta se automatski menja na false.
Sto se tiče izgleda formulara, nikakve dalje intervencije nisu potrebne, ali moramo izmeniti kod. Prvo moramo izbrisati sve što se odnosi na okvir za tekst koji smo uklonili. Idite na kod i uradite sledeće:
6. U konstruktoru formulara izbrišite redove koje se odnose na **txtZanimanje**. Ovde se misli na prijavljivanje događaja **Validacija** i **TextPromenjen**, kao i red koji podešava svojstvo **tag** kontrole **txtZanimanje** na false.
7. U potpunosti uklonite metodu **txtZanimanje_Validacija**.

Dodavanje rutina za obradu događaja

Metod **txtBox_TextPromenjen** sadržao je test u kome smo proveravali da li je pozivajuća kontrola bila okvir za tekst txtOccupation. Za sada sigurno znamo da nije; zato ćemo promeniti metodu brisanjem bloka **else if** i promenom naredbe **if,** tako da kod treba da izgleda:

```
389     private void txtBox_TextPromenjen(object sender, System.EventArgs e)
390     {
391       // Konvertujemo parametar sender u tip TextBox
392       TextBox tb = (TextBox)sender;
393
394       // Proveravamo ispravnost podataka i
395       // podesavamo boju boju pozadine.
396       if (tb.Text.Length == 0)
397       {
398         tb.Tag = false;
399         tb.BackColor = Color.Red;
400       }
401       else
402       {
403         tb.Tag = true;
404         tb.BackColor = SystemColors.Window;
405       }
406
407       // Podesavamo dugme OK
408       ValidacijaZaSve();
409     }
```

Na još jednom mestu radimo proveru vrednosti tekstualnog okvira koji smo uklonili - u metodi ValidateAll (). Potpuno izbrišite postojeće provere i unesite sledeći kod:

```
372     private void ValidacijaZaSve()
373     {
374       // Setovanje osobine Enabled dugmeta OK
375       // na true ako su svi Tag vrednosti true.
376       this.btnOK.Enabled = ((bool)(this.txtAdresa.Tag) &&
377         (bool)(this.txtGodiste.Tag) &&
378         (bool)(this.txtIme.Tag));
379     }
```

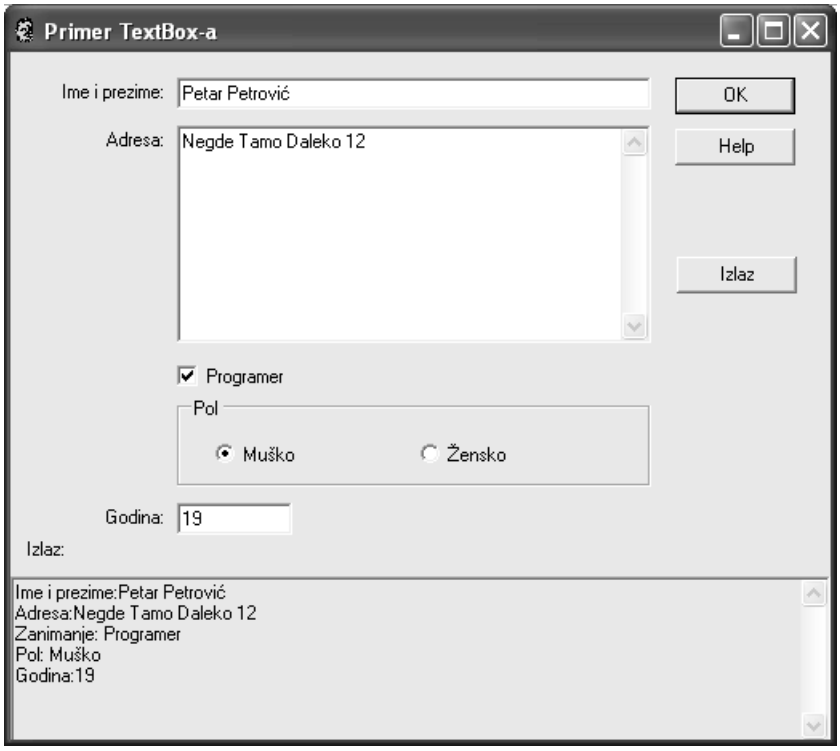
Pošto koristimo polje za potvrdu, umesto okvira za tekst, znamo da korisnik ne može uneti nekorektan podatak, jer će on ili ona uvek biti ili programer, ili neće biti programer.
Takođe, znamo da je korisnik ili muško ili žensko. A pošto smo podesili osobinu jednog radio dugmeta na true, nemoguće je da korisnik izabere neispravan podatak. Stoga, jedino što preostaje da se uradi jeste izmena teksta u kome nalazimo pomoć, i izlaznog teksta. Ovo radimo u rutini za obradu događaja dugmeta:

```
330     private void btnHelp_Click(object sender, System.EventArgs e)
331     {
332       // Pisemo kratak opis svakog textbox-a u izlaznom textbox-u
333       string izlaz;
334       izlaz = "Ime i prezime = Tvoje ime i prezime\r\n";
335       izlaz += "Adresa = Tvoja adresa\r\n";
336       izlaz += "Programer = Cekiraj te 'Programer' ako ste programer\r\n";
337       izlaz += "Pol = Izaberite vaš pol\r\n";
338       izlaz += "Godina = Godina rodenja";
339       //Ubacimo novi tekst
340       this.txtIzlaz.Text = izlaz;
341     }
```

U metodi za dugme OK promene su malo interesantnije:

```
313     private void btnOK_Click(object sender, System.EventArgs e)
314     {
315       // Nema provere ispravnosti, pošto nije ni potrebna
316       string izlaz;
317       //Spajamo tekstove cetiri textbox-a
318       izlaz = "Ime i prezime:" + this.txtIme.Text + "\r\n";
319       izlaz += "Adresa:" + this.txtAdresa.Text + "\r\n";
320
321       izlaz += "Zanimanje: " + (string)(this.chkPrograme.Checked ?
322                   "Programer" : "Not a programmer") + "\r\n";
323       izlaz += "Pol: " + (string)(this.rdoZensko.Checked ?
324                   "Žensko" : "Muško") + "\r\n";
325       izlaz += "Godina:" + this.txtGodiste.Text;
326       // Ubacujemo novi tekst
327       this.txtIzlaz.Text = izlaz;
328     }
```

Prvi od novih redova, koji je naglašen, prikazuje zanimanje korisnika. Proveravamo svojstvo Checked polja za potvrdu; ako je podešeno na true, prikazujemo niz znakova Programer. A ako je podešeno na false, prikazujemo **Nije programer**.
Drugi red proverava samo radio dugme **rdoZensko**. Ako je svojstvo Checked podešeno na true, znamo da je korisnik žensko. Ako je podešeno na false, korisnik je muško. Pošto postoje samo dve moguće opcije, nema potrebe da proveravamo drugo radio dugme, jer će svojstvo Checked tog radio dugmeta uvek biti suprotno podešeno od svojstva Checked prvog radio dugmeta.
Da smo imali vise od dva radio dugmeta, morali bismo proći kroz sve od njih, dok ne pronađemo ono kod kog je svojstvo Checked podešeno na true.
Kada pokrenete ovaj primer, trebalo bi da dobijete nešto slično ovome:



Kontrola RichTextBox

Kao i običan okvir za tekst, kontrola RichTextBox (obogaćen okvir za tekst) izvedena je iz klase TextBoxBase. Zbog toga ovi okviri imaju neke sličnosti, ali je kontrola RichTextBox mnogo sadržajnija. Okvir za tekst koristimo za unos kratkih nizova znakova, dok obogaćeni tekstualni okvir koristimo za prikazivanje i unos formatiranog teksta (npr. polucrno, podvučeno ili kurziv). Ovo postižemo korišćenjem standarda za formatirani tekst koji se zove Rich Text Format (obogaćen tekstualni format) ili RTF.

U prethodnom primeru, koristili smo standardni okvir za tekst. Mogli smo umesto toga koristiti i obogaćeni okvir za tekst. Kao što ćete videti u primeru koji sledi, možete ukloniti okvir za tekst pod nazivom txtOutput, i na njegovo mesto ubaciti obogaćeni okvir za tekst sa istim imenom. Program će se ponašati isto kao i ranije.

Svojstva kontrole RichTextBox

S obzirom na prednosti obogaćenog okvira za tekst, on ima i neke nove osobine. Sledeća tabela opisuje neke najčešće korišćene osobine:

Ime osobine	Dostupnost	Opis
CanRedo	Samo čitanje	Vrednost ove osobine je true, ako nešto što je poništeno može biti ponovljeno; u suprotnom je false.
CanUndo	Samo čitanje	Vrednost ove osobine je true, ako je moguće izvršiti akciju Undo na obogaćeni okvir za tekst; u suprotnom je false.
RedoAc t i onName	Samo čitanje	Ovo svojstvo sadrži ime akcije koja će biti korišćena da bi se ponovo uradila neka prethodno poništena radnja u obogaćenom okvira za tekst.
DetectUrls	Čitanje/upis	Podesite ovu osobinu na true ako želite da kontrola detektuje URL adrese i da ih formatira (podvuče, kao u čitaču).
Rtf	Čitanje/upis	Ovo odgovara svojstvu Text, s tim što je tekst u RTF formatu.
SelectedRtf	Čitanje/upis	Koristite ovu osobinu da biste dobili ili podesili izabrani tekst na RTF format. Tekst ovako kopiran u drugu aplikaciju, npr. MS Word, zadržava svoj format.
SelectedText	Čitanje/upis	Kao i SelectedRtf, možete dobiti ili podesiti izabrani tekst. Za razliku od prethodne osobine, pri kopiranju tekst ne zadržava svoj format.
SelectionAlignment	Čitanje/upis	Ovo svojstvo predstavlja način poravnavanja izabranog teksta. Može biti Center, Left ili Right (centar, levo ili desno).
SelectionBullet	Čitanje/upis	Uz pomoć ove osobine možete saznati da li je neki deo teksta formatiran sa simbolom za nabrajanje, a možete umetati ili uklanjati simbol za nabrajanje.
Bulletlndent	Čitanje/upis	Koristite ovu osobinu da odredite za koliko piksela treba uvući simbol za nabrajanje.
SelectionColor	Čitanje/upis	Dopušta vam da menjate boju izabranog teksta.
SelectionFont	Čitanje/upis	Dopušta vam da menjate font izabranog teksta.
SelectionLenght	Čitanje/upis	Uz pomoć ove osobine možete podesiti ili saznati dužinu izabranog teksta.
SelectionType	Samo čitanje	Ovo svojstvo sadrži informacije o izabranom delu. Red će vam da li je izabran neki OLE objekat ili je izabran samo tekst.
ShowSelectionMargin	Čitanje/upis	Ako podesite ovo svojstvo na true, margina će se pojaviti na levoj strani kontrole RichTextBox. Ovo omogućava lakše biranje teksta.
UndoAc t i onName	Samo čitanje	Uzima ime akcije koja će se koristi u ako korisnik želi da poništi neku radnju.
SelectionProtected	Čitanje/upis	Ako podesite ovu osobinu na true, možete odrediti da se određeni deo teksta ne menja.

Kao što možete videti, većina novih svojstava se odnosi na biranje. To je zato što se formatiranje odnosi na onaj deo teksta koji je korisnik izabrao. Ukoliko nikakav izbor nije napravljen, formatiranje će započeti od tačke u tekstu gde je postavljen kursor, a koja se naziva tačka umetanja.

Događaji kontrole RichTextBox

Većina događaja koji se koriste u obogaćenim okvirima za tekst, isti su kao i za obične okvire za tekst. Ima i nekoliko novih:

Ime osobine	Opis
LinkedClick	Ovaj događaj se šalje kada korisnik pritisne vezu unutar teksta.
Protected	Ovaj događaj se šalje kada korisnik pokuša da promeni tekst koji je označen kao zaštićen.
SelctionChanged	Ovaj događaj se šalje kada želite nešto da izmenite u izabranom tekstu. Ako iz bilo kojih razloga ne želite da dopustite korisniku da menja izabrani tekst, možete da ga sprečite pomoću ovog događaja.

Vežba br. 24.

U ovom primeru kreiraćemo krajnje jednostavan program za uređivanje teksta, koji pokazuje kako da izmenite osnovno formatiranja teksta i kako da učitate i snimite tekst iz obogaćenog okvira za tekst. Zbog jednostavnosti, učitavaćemo iz fiksirane datoteke i snimaćemo u nju.

Napravite novi projekat konzolne aplikacije birajući File | New | Project... iz menija:

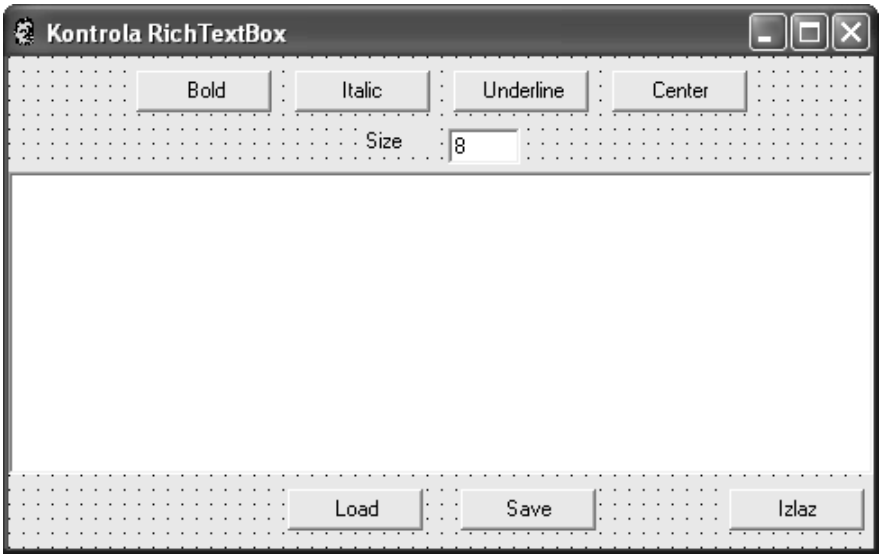
Izaberite Visual C# Projects direktorijum unutar prozora Project Types:, i tip projekta Windows Application u okviru prozora Templates: U okviru za tekst Location: promenite putanju u **C:\Temp\SoftIng\LecturesCode\Vezba24** (ovaj će direktorijum biti automatski napravljen ukoliko već ne postoji), i ostavite podrazumevani tekst u okviru za tekst Name: **Kontrola RichTextBox.**

Otvorite paletu Properties i za kontrolu *forme* podesite sledeće osobine:

Osobina (Properties)	Vrednost (Value)
Name	fclsMain
Text	Kontrola RichTextBox
StartPosition	CenterScreen
Font	Microsoft Sans Serif, Regular, 8, Central European
FormBorderStyle	FixedSingle
Size	486; 305

Na formu postavite 7 kontroli tipa **Button**, jednu kontrolu **Label**, jednu kontrolu **TextBox** i jednu kontrolu tipa **RichTextBox**. Podesite njihova svojstva redom prikazano u tabelama.

Osobina (Properties)	Vrednost (Value) za kontrolu						
	Button1	**Button2**	**Button3**	**Button4**	**Button5**	**Button6**	**Button7**
Name	btnBold	btnItalic	btnUnderline	btnCentar	bthLoad	btnSave	btnIzlaz
Text	&Bold	&Italic	&Underline	&Centar	&Load	&Save	&Izlaz
Cursor	Hand						
Location	71; 8	159; 8	247; 8	335; 8	155; 240	251; 240	400; 240
Size	75; 23						



za TextBox1

Osobina (Properties)	Vrednost (Value)
Name	txtVelicinaFonta
Text	8
Location	244; 40
Size	40; 20

Na formu postavite kontrolu tipa **RichTextBox.** I postavite svojstva kao u sledećoj tabeli.

Osobina (Properties)	Vrednost (Value)
Name	rtfText
Location	0; 64
Size	480; 168

Podesite svojstva Anchor kontrola kao u tabeli:

Ime kontrole	Anchor vrednosti
btnLoad i btnSave	Buttom
rtfText	Top, Left, Bottom, Right
Sve ostale	Top

Podesite svojstvo MinimumSize formulara da bude isto kao svojstvo Size.

Dodavanje rutina za obradu događaja

Ovim smo završili sa izgledom formulara i prelazimo na kod. Dvaput pritisnite na dugme Bold da biste dodali ratinu za obradu za Click događaj. Ovako izgleda kod:

```
private void btnBold_Click(object sender, System.EventArgs e)
{
    Font stariFont;
    Font noviFont;

    // Dobijamo font koji se trenutno koristi u izabranom tekstu
    stariFont = this.rtfText.SelectionFont;

    // Ako se koristi bold font, onda ga uklonimo
    if (stariFont.Bold)
        noviFont = new Font(stariFont, stariFont.Style & ~FontStyle.Bold);
    else
        noviFont = new Font(stariFont, stariFont.Style | FontStyle.Bold);

    // Ubacujemo novi font i vratimo fokus na RichTextBox kontrolu
    this.rtfText.SelectionFont = noviFont;
    this.rtfText.Focus();
}
```

Počinjemo proverom fonta koji je korišćen u trenutno izabranom tekstu i dodelimo taj font lokalnoj promenljivoj. Zatim, proverimo da li je izabrani tekst već formatiran kao polucrn. Ako jeste, onda to vise neće biti; u suprotnom želimo da podesimo da bude polucrn. Kreiramo novi font koristeći oldFont kao prototip, a dodajemo ili uklanjamo polucrni stil, zavisno od potrebe. Na kraju, dodeljujemo novi font izabranom tekstu i vraćamo fokus na kontrolu RichTextBox.

Rutine za obradu za btnltalic i btnUnderline su slime kao i prethodne, s tim što proveravamo odgovarajuće stilove. Dva puta pritisnite na dugmad Italic i Underline i dodajte sledeći kod:

```
private void btnItalic_Click(object sender, System.EventArgs e)
{
    Font stariFont;
    Font noviFont;


    // Dobijamo font koji se trenutno koristi u izabranom tekstu
    stariFont = this.rtfText.SelectionFont;

    // Ako je stil koji se koristi je kurziv, onda ga uklonjamo
    if (this.rtfText.SelectionFont.Italic)
        noviFont = new Font(stariFont, stariFont.Style & ~FontStyle.Italic);
    else
        noviFont = new Font(stariFont, stariFont.Style | FontStyle.Italic);

    // Ubacujemo novi font i vratimo fokus na RichTextBox kontrolu
    this.rtfText.SelectionFont = noviFont;
    this.rtfText.Focus();
}
```

```
247         private void btnUnderline_Click(object sender, System.EventArgs e)
248         {
249             Font stariFont;
250             Font noviFont;
251 
252             // Dobijamo font koji se trenutno koristi u izabranom tekstu
253             stariFont = this.rtfText.SelectionFont;
254 
255             // Ako je stil koji se koristi je podvucen, onda ga uklonjamo
256             if (this.rtfText.SelectionFont.Underline)
257                 noviFont = new Font(stariFont, stariFont.Style & ~FontStyle.Underline);
258             else
259                 noviFont = new Font(stariFont, stariFont.Style | FontStyle.Underline);
260 
261             // Ubacujemo novi font i vratimo fokus na RichTextBox kontrolu
262             this.rtfText.SelectionFont = noviFont;
263             this.rtfText.Focus();
264         }
```

Dvaput pritisnite na poslednje dugme za formiranje, Center, i dodajte sledeći kod:

```
266         private void btnCenter_Click(object sender, System.EventArgs e)
267         {
268             if (this.rtfText.SelectionAlignment == HorizontalAlignment.Center)
269                 this.rtfText.SelectionAlignment = HorizontalAlignment.Left;
270             else
271                 this.rtfText.SelectionAlignment = HorizontalAlignment.Center;
272             this.rtfText.Focus();
273         }
```

Ovde moramo proveriti još jednu osobinu, SelectionAlignment, da bismo videli da li je izabrani tekst već centriran. HorizontalAlignment može uzimati vrednosti iz enumeracije, koje mogu biti Left, Right, Center, Justify ili NotSet. U ovom slučaju jednostavno proverimo da li je podešeno na Center, i ako jeste, poravnamo tekst po levoj ivici. A ako nije, podesimo tekst na centar.
Poslednje formatiranje koje ćemo uraditi jeste podešavanje veličine teksta. Dodajemo dve ratine za obradu događaja za tekstualni okvir Size: jedan za kontrolu unosa i drugi koji detektuje da li je korisnik završio unos podatka.

Dodajte sledeći kod u konstruktor formulara:

```
34          public fclsMain()
35          {
36              InitializeComponent();
37 
38              // Prijava dogadjaja
39              this.txtVelicinaFonta.KeyPress += new
40                  System.Windows.Forms.KeyPressEventHandler(this.txtSize_KeyPress);
41              txtVelicinaFonta.Validating += new
42                  System.ComponentModel.CancelEventHandler(this.txtSize_Validacija);
```

Ove dve rutine za obradu videli smo u prethodnom primeru. Obe koriste pomoćnu metodu ApplyTextSize, koja uzima string sa veličinom teksta:

```
278         private void txtSize_KeyPress(object sender,
279             System.Windows.Forms.KeyPressEventArgs e)
280         {
281             // Uklanjama sve karaktere koji nisu brojevi, backspace ili enter
282             if ((e.KeyChar < 48 || e.KeyChar > 57) && e.KeyChar != 8 && e.KeyChar != 13)
283             {
284                 e.Handled = true;
285             }
286             else if (e.KeyChar == 13)
287             {
288                 // Primenjujemo velecinu ako korisnik pritisne enter
289                 TextBox txt = (TextBox)sender;
290 
291                 if (txt.Text.Length > 0)
292                     PrihvatiTextSize(txt.Text);
293                 e.Handled = true;
294                 this.rtfText.Focus();
295             }
296         }
```

```
298         private void txtSize_Validacija(object sender,
299             System.ComponentModel.CancelEventArgs e)
300         {
301             TextBox txt = (TextBox)sender;
302 
303             PrihvatiTextSize(txt.Text);
304             this.rtfText.Focus();
305         }
306 
307         private void PrihvatiTextSize(string textSize)
308         {
309             //Konvertujemo tekst u float
310             float newSize = Convert.ToSingle(textSize);
311             FontFamily currentFontFamily;
312             Font newFont;
313 
314             // Konvertujemo novi font iste familije, ali razlicite velicine
315             currentFontFamily = this.rtfText.SelectionFont.FontFamily;
316             newFont = new Font(currentFontFamily, newSize);
317 
318             // Podesimo font izabranog teksta na novi font
319             this.rtfText.SelectionFont = newFont;
320         }
```

Ono što nas zanima dešava se u pomoćnoj metodi ApplyTextSize. Ova metoda počinje konverzijom stringa (unetog u Size polje) u tip float. Sprečili smo korisnika da unosi bilo šta drugo osim celih brojeva; međutim, kada kreiramo novi font, potreban nam je tip float - zato radimo ovu konverziju tipova.
Posle toga, koristimo familiju fontova kojoj pripada font, kreiramo novi font u okviru te iste familije, ali sa novom veličinom. Na kraju, podesimo font izabranog teksta na novokreirani font.
To je sve što možemo da uradimo što se tiče formatiranja, iako neke stvari kontrola RichTextBox sama rešava. Ako pokrenete ovaj primer, možete podesiti tekst na polucrn, kurziv ili podvučen, i možete da centrirate tekst. Ovo je očekivano, ali ima još nešto interesantno. Probajte da otkucate Web adresu u tekstu, npr. **www.tf.zr.ac.yu**. Kontrola ovo prepoznaje kao internet adresu, podvlači je, a kada se pokazivač miša pomeri na adresu, pretvara se u ruku. Ako mislite da možete da je pritisnete i da vam se otvori odgovarajuća Web stranica, skoro da ste u pravu. Moramo se pozabaviti događajem koji se šalje kada korisnik pritisne na hipervezu: to je događaj LinkClicked.

Ovo ćemo uraditi prijavom događaja u konstruktoru, kao što smo navikli:

```
44              this.rtfText.LinkClicked += new
45                  System.Windows.Forms.LinkClickedEventHandler(this.rtfText_LinkedClick);
46
47
```

Ovu rutinu za obradu događaja nismo videli do sada. Koristi se da se dođe do teksta veze koja je pritisnuta. Rutina je iznenađujuće jednostavna:

```
322         private void rtfText_LinkedClick(object sender,
323                                 System.Windows.Forms.LinkClickedEventArgs e)
324         {
325             System.Diagnostics.Process.Start(e.LinkText);
326         }
```

Ovaj kod otvara podrazumevani čitač, ako prethodno nije otvoren, i vodi nas na lokaciju na koju upućuje hiperveza.

Deo aplikacije kojim se uređuje tekst je završen. Sve što preostaje jeste učitavanje i snimanje sadržaja kontrole. Za to koristimo fiksiranu datoteku.

Dva puta pritisnite na dugme Load i dodajte sledeći kod:

```
328         private void btnLoad_Click(object sender, System.EventArgs e)
329         {
330             // Ucitavamo tekst u kontrolu RichTextBox
331             try
332             {
333                 rtfText.LoadFile(@"../../Test.rtf");
334             }
335             catch (System.IO.FileNotFoundException)
336             {
337                 MessageBox.Show("Jos nije ucitan fajl");
338             }
339         }
340
```

Ništa vise nije potrebno. Pošto radimo sa datotekama, uvek se mogu pojaviti posebni slučajevi, koje moramo da rešimo. U metodi Load moramo rešiti poseban slučaj koji se dešava ako datoteka ne postoji. To je jednostavno kao i snimanje datoteka. Dvaput pritisnite na dugme Save i dodajte sledeći kod:

```
341         private void btnSave_Click(object sender, System.EventArgs e)
342         {
343             // Snimamo tekst
344             try
345             {
346                 rtfText.SaveFile(@"../../Test.rtf");
347             }
348             catch (System.Exception err)
349             {
350                 MessageBox.Show(err.Message);
351             }
352         }
```

Pokrenite ponovo program, formatirajte tekst i pritisnite dugme Save. Izbrišite sadržaj okvira, a zatim pritisnite Load i pojaviće se tekst koji ste upravo snimili.

Kontrole ListBox i CheckedListBox

Okvir sa listom koristi se za prikaz liste stringova od kojih jedan ili vise može biti odjednom izabrano. Kao kod polja za potvrdu i radio dugmeta, okvir sa listom traži od korisnika da izabere jednu ili vise mogućih opcija. Okvir liste trebalo bi da koristite kada, u toku kreiranja aplikacije, ne znate tačan broj opcija koje korisnik može izabrati (npr. lista zaposlenih). Čak i kada unapred znate broj mogućih konkretnih opcija, trebalo bi razmisliti o upotrebi okvira liste ako je taj broj veliki.

Klasa ListBox je izvedena iz klase ListControl, koja obezbeđuje osnovno funkcionisanje dve kontrole sa listama koje su sastavni deo Visual Studio .NET-a. O drugoj kontroli, ComboBox, pričaćemo kasnije u ovom predavanju.

Još jedna vrsta okvira liste dolazi sa Visual Studio .NET-om: CheckedListBox koja je izvedena iz klase ListBox. Ova kontrola omogućava potvrdu pojedinačnog stringa iz liste.

Svojstva kontrole ListBox

Sve osobine opisane u tabeli koja sledi važe i za ListBox i za CheckedListBox, osim ako nije drugačije naznačeno:

Ime osobine	Dostupnost	Opis
SelectedIndex	Čitanje/upis	Vrednost koja pokazuje indeks izabranog elementa u okvira liste (prvi element u listi ima indeks 0). Ako okvir liste dopušta da vise elemenata bude izabrano odjednom, ovo svojstvo uzima indeks prvog izabranog elementa.
ColumnWidth	Čitanje/upis	U okvirima sa vise kolona, ovo svojstvo određuje širinu kolona.
Items	Samo čitanje	Kolekcija Items sadrži sve elemente liste. Koristite ovu osobinu za dodavanje ili izbacivanje elementa.
MultiColumn	Čitanje/upis	U okvirima sa vise kolona, ovo svojstvo određuje širinu kolona.
SelectedIndices	Samo čitanje	Ovo svojstvo je kolekcija koja sadrži indekse izabranih elemenata, pri čemu indeksiranje počinje od nule.
SelectedItem	Čitanje/upis	U listi koja dopušta biranje samo jednog elementa, ovo svojstvo sadrži taj element. Ako se u listi može izabrati vise elemenata, svojstvo sadrži prvi izabrani element.
SelectedItems	Samo čitanje	Ovo svojstvo je kolekcija koja sadrži sve trenutno izabrane elemente.
SelectionMode	Čitanje/upis	Postoje četiri načina biranja elemenata u listi: • None: nijedan element ne može biti izabran. • One: samo jedan element može biti izabran. • MultiSimple: vise elemenata može biti izabrano. • MultiExtended: vise elemenata može biti izabrano i može se koristiti Ctrl, Shift i tasteri sa strelicama za biranje.
Sorted	Čitanje/upis	Ako podesite ovu osobinu na true, kontrola ListBoxce sortirati elemente po abecednom redu.

Ime osobine	Dostupnost	Opis
Text	Čitanje/upis	Ovo svojstvo se razlikuje od Text svojstava ostalih kontrola. Ako podesite Text osobinu okvira liste, ona će tragati za elementom koji ima to ime i izabraće ga. Ako, sa druge strane, čitate vrednost Text osobine, vraćena vrednost je prvi izabrani element. Ovo svojstvo ne može biti korišćeno ako je svojstvo SelectionMode podešeno na None.
Checkedlndicies	Samo čitanje	Samo za kontrolu CheckedListBox: ovo svojstvo sadrži sve indekse elemenata koji su potvrđeni ili koji imaju neodređen status.
CheckedItems	Samo čitanje	Samo za kontrolu CheckedLi s tBox: ovo je kolekcija koja sadrži sve elemente koji su potvrđeni ili koji imaju neodređen status.
CheckOnClick	Čitanje/upis	Samo za kontrolu CheckedListBox: ako je ovo svojstvo podešeno na true, element menja stanje kad god korisnik pritisne na njega.
ThreeDCheckBoxes	Čitanje/upis	Samo za kontrolu CheckedListBox: podešavanjem ove osobine možete birati da li će polja za potvrdu biti ravna ili normalna.

Metode kontrole ListBox

Zbog efikasnog rada sa okvirima liste, morate znati neke metode koje okvir može pozvati. Tabela koja sledi opisuje samo najčešće korišćene metode. Metode važe i za kontrolu ListBox i za CheckedListBox, osim ako nije drugačije naznačeno:

Ime	Opis
ClearSelected	Poništava izbor u okviru liste.
FindString	Pronalazi prvi string u listi koji počinje željenim stringom, npr. FindString ("a ") će naći prvi string u listi koji počinje sa 'a'.
FindstringExact	Kao i FindString, s dm što se kompletan string mora podudarati.
GetSelected	Vraća vrednost koja pokazuje da li je element izabran ili nije.
SetSelected	Podešava da element bude izabran, ili briše element iz liste izabranih elemenata.
ToString	Vraća trenutno izabrani element.
GetltemChecked	Samo za kontrolu CheckedLi stBox: vraća vrednost koja pokazuje da li je element potvrđen ili ne.
GetltemCheckState	Samo za kontrolu CheckedListBox: vraća vrednost koja pokazuje stanje elementa.
SetltemChecked	Samo za kontrolu CheckedListBox: podešava određeni element tako da bude potvrđen
SetltemCheckState	Samo za kontrolu CheckedListBox: podešava stanje elementa.

Događaji kontrole ListBox

Obično, događaji kojih želite da budete svesni dok radite sa kontrolama ListBox i CheckedListBox, jesu oni događaji koji se odnose na izbor korisnika:

Ime	Opis
ItemCheck	Samo za kontrolu CheckedListBox: dešava se kada se menja stanje elementa.
SelectedIndexChanged	Dešava se kada se menja indeks izabranog elementa.

Primer kontrole ListBox

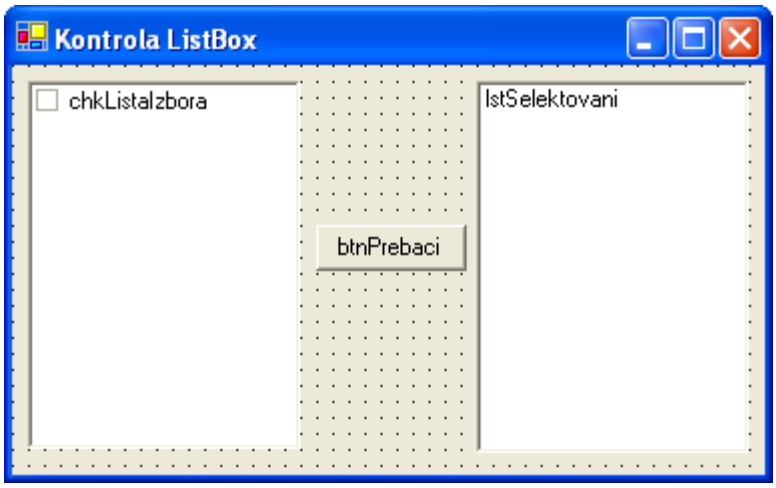
Vežba br. 25.

Napravićemo manji primer koji uključuje kontrole ListBox i CheckedListBox. Korisnik može potvrditi element u kontroli CheckedListBox, zatim pritisnuti na dugme koje će pomeriti potvrđene elemente u kontrolu ListBox. Kreiramo dijalog na sledeći način:

Napravite novi projekat konzolne aplikacije birajući File | New | Project... iz menija:

Izaberite Visual C# Projects direktorijum unutar prozora Project Types: i tip projekta Windows Application u okviru prozora Templates: U okviru za tekst Location: promenite putanju u **C:\Temp\SoftIng\LecturesCode\Vezba25** (ovaj će direktorijum biti automatski napravljen ukoliko već ne postoji), i ostavite podrazumevani tekst u okviru za tekst Name: **Kontrola ListBox.**

Dodajte kontrole ListBox, CheckedListBox i Button, i imenujte ih kao što je prikazano na slici:



Otvorite paletu Properties i za kontrolu *forme* podesite sledeće osobine:

Osobina (Properties)	Vrednost (Value)
Name	fclsMain
Text	Kontrola ListBox
StartPosition	CenterScreen
Font	Microsoft Sans Serif, Regular, 8, Central European
FormBorderStyle	FixedSingle
Size	382; 237

za Button1

Osobina (Properties)	Vrednost (Value)
Name	btnPrebaci
Text	Prebaci
Location	152; 80
Size	75; 23

za ChekedListBox1

Osobina (Properties)	Vrednost (Value)
Name	chkListaIzbora
CheckOnClick	True
Items	jedan dva tri četiri pet šest sedam osam devet
Location	8; 8
Size	136; 184

za ListBox1

Osobina (Properties)	Vrednost (Value)
Name	lstSelektovani
Location	232; 8
Size	136; 186

Dodavanje rutina za obradu događaja

Sada smo spremni za dodavanje koda. Kada korisnik pritisne dugme Move, želimo da pronademo elemente koji su potvrđeni i da ih kopiramo u okvir sa listom Selected.
Dva puta pritisnite dugme i dodajte sledeći kod:

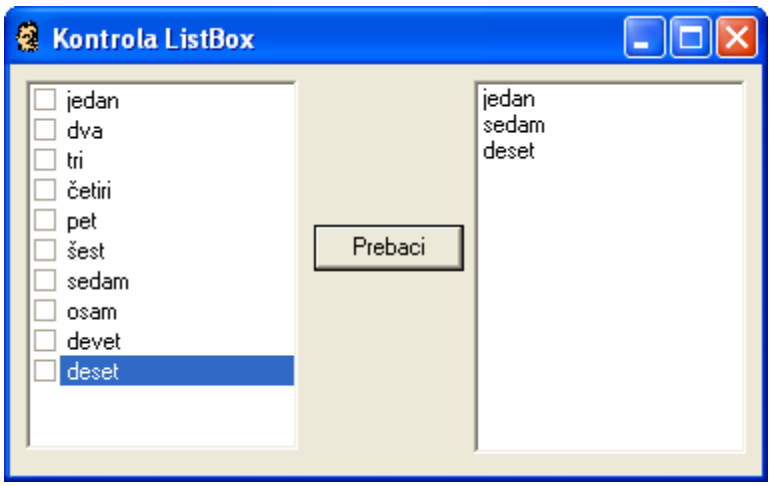
```
127    private void btnMove_Click(object sender, System.EventArgs e)
128    {
129      // Proveravamo postoji li neki potvrđeni element u kontroli CheckedListBox
130      if (this.chkListaIzbora.CheckedItems.Count > 0)
131      {
132        // Ispraznimo kontrolu ListBox u koju stavljamo izabrane elemnt
133        this.lstSelektovani.Items.Clear();
134
135        // Prođimo u petlji kroz kolekciju CheckedItems kontrole CheskedListBox
136        // i dodajemo elemnte u okvir Selected
137        foreach (string item in this.chkListaIzbora.CheckedItems)
138        {
139          this.lstSelektovani.Items.Add(item.ToString());
140        }
141
142        // Poništimo sve što je potvrđeno u kontroli CheskedListBox
143        for (int i = 0; i < this.chkListaIzbora.Items.Count; i++)
144          this.chkListaIzbora.SetItemChecked(i, false);
145      }
```

Počinjemo tako što proveravamo svojstvo **Count** kolekcije **CheckedItems**. Vrednost osobine Count biće veća od nule ako je neki element liste potvrđen. Zatim, uklanjamo sve elemente iz okvira Selected, prolazimo kroz kolekciju **CheckedItems** dodajući svaki element te kolekcije u okvir Selected. Na kraju, podešavamo sve elemente u okvira CheckedListBox tako da ne budu potvrđeni.

Sada nam još trebaju elementi koje ćemo pomerati. Mogli bismo dodati ove elemente u režimu dizajna, biranjem svojstva **Item** i upisivanjem stavki. Umesto toga, dodaćemo elemente direktno iz koda. To postižemo u konstruktoru formulara:

```
26
27      public fclsMain()
28      {
29        //
30        // Required for Windows Form Designer support
31        //
32        InitializeComponent();
33
34        // Dodajemo još jednu stavku u kontroli chkListaIzbora
35        this.chkListaIzbora.Items.Add("deset");
36      }
37
```

Kao što biste unosili vrednosti u panelu Properties, možete koristiti kolekciju Items u vreme izvršavanja.
Ovim završavamo primer za okvire liste. Ako pokrenete ovaj program, dobićete nešto nalik ovome:



Korišćenje kontrola za Windows formulare – III deo

Kontrola ComboBox

što ime implicira, kontrola ComboBox (kombinovani okvir) sadrži nekoliko kontrola, tačnije: kontrole TextBox, Button i ListBox. Za razliku od kontrole Listbox, u kontroli ComboBox nije moguće izabrati vise od jednog elementa liste, ali je moguće uneti nove elemente otkucavanjem njihovih imena u polju TextBox kombinovanog okvira.
Uobičajeno je da se kontrola ComboBox koristi kada želimo da uštedimo prostor u dijalogu, zbog toga što su stalno vidljivi delovi kombinovanog okvira samo tekstualni okvir i kontrole Button. Kada korisnik pritisne na dugme sa strelicom desno od tekstualnog okvira, pojavljuje se okvir liste u kojem korisnik može izabrati određeni element. Čim korisnik izabere neki element, okvir liste nestaje, a izabrani element se pojavljuje u tekstualnom okvira, vraćajući normalan izgled dijalogu. Pogledaćemo osobine i događaje kontrole ComboBox i napraviti primer koji uključuje ComboBox i dve kontrole ListBox.

Svojstva kontrole ComboBox

Kako kontrola ComboBox poseduje mogućnosti kontrola TextBox i ListBox, mnoge osobine kombinovanog okvira mogu se naći i u dve pomenute kontrole; zbog ovoga, postoji ogroman broj svojstava i događaja kontrole ComboBox, ali mi demo opisati samo najčešće korišćene. Kompletnu listu pogledajte u **MSDN biblioteci.**

Ime osobine	Dostupnost	Opis
DropDownStyle	Čitanje/upis	Kombinovani okvir može biti prikazan u tri različita stila: • DropDown: korisnik može upisivati u tekstualni okvir, ali mora pritisnuti na dugme sa strelicom za prikaz kompletne liste elemenata. • Simple: isto kao i DropDown s dm što je lista uvek vidljiva, slično običnoj kontroli ListBox. • DropDownList: korisnik ne može upisivati novi element u tekstualni okvir i mora pritisnuti na dugme sa strelicom za prikaz liste elemenata.
DroppedDown	Čitanje/upis	Pokazuje da li je lista prikazana ili nije. Ako podesite ovu osobinu na true, lista će se prikazati.
Items	Samo čitanje	Kolekcija koja sadrži sve elemente liste iz kombinovanog okvira.
MaxLength	Čitanje/upis	Podešavanjem ove osobine na broj veći od nule, određujete maksimalan broj karaktera koji je moguće uneti u tekstualni deo kontrole.
SelectedIndex	Čitanje/upis	Pokazuje indeks trenutno izabranog elementa liste.
SelectedItem	Čitanje/upis	Pokazuje trenutno izabrani element liste.
SelectedText	Čitanje/upis	Predstavlja tekst koji je izabran u tekstualnom delu kontrole.
SelectionStart	Čitanje/upis	Predstavlja indeks prvog karaktera dela teksta koji je izabran u tekstualnom delu kontrole.
SelectionLength	Čitanje/upis	Predstavlja dužinu izabranog teksta u tekstualnom delu kontrole.
Sorted	Čitanje/upis	Ako podesite ovu osobinu na true, kontrola će sortirati elemente liste po abecednom redu.
Text	Čitanje/upis	Ako podesite ovu osobinu na vrednost null, ništa neće biti izabrano u lisa. Ako podesite na vrednost koja postoji u listi, ta vrednost će biti izabrana. Ako tražena vrednost ne postoji u listi, tekst će biti prikazan u tekstualnom delu kontrole

Događaji kontrole ComboBox

Postoje tri osnovne akcije za koje želite da znate kada se dese u kombinovanom okviru:

- Kada se promeni izbor;
- Kada se menja stanje liste (da li je prikazana ili nije);
- Kada se tekst menja.

Da biste obradili ove tri akcije, obično se morate prijaviti za jedan ili više od sledećih događaja:

Ime	Opis
DropDown	Dešava se kada se prikaže deo sa okvirom sa listom.
SelectedlndexChanged	Dešava se kada se umesto jednog elementa u listi izabere neki drugi element.
KeyDown KeyPress KeyUp	Ovi događaji dešavaju se kada se pritisne taster dok tekstualni deo kontrole ima fokus. Pogledajte objašnjenja ovih događaja za tekstualni okvir.
TextChanged	Dešava se kada se menja svojstvo Text.

Primer kontrole ComboBox

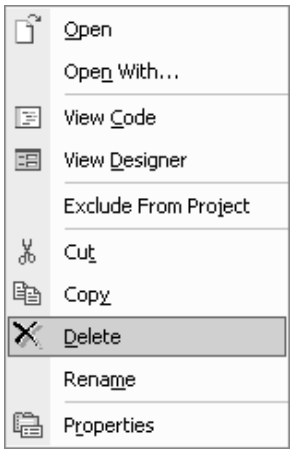
Vežba br. 26.

Ponovo ćemo koristiti primer za tekstualne okvire. Setite se da smo u primeru za kontrolu CheckBox promenili okvir za tekst **Zanimanje** u polje za potvrdu. Trebalo bi da postoji neko drago zanimanje osim programera, zato ćemo napraviti još jednu izmenu. Za ovu promenu koristićemo kombinovani okvir koji će sadržati dva moguća zanimanja: konsultant i programer.

Takođe, dopustićemo korisniku da unese i ostala zanimanja, ukoliko korisnik nije ni konsultant ni programer. Da korisnik ne bi morao iznova unositi svoje zanimanje svaki put kada koristi aplikaciju, snimićemo uneti element u datoteku kad god se dijalog zatvara. Takođe, učitaćemo sve elemente iz datoteke kad god se dijalog otvara.

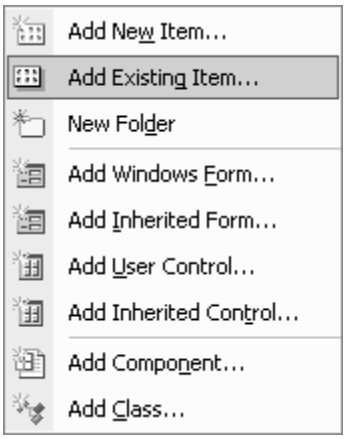
Stari projekat odnosno vežbu pod brojem 22. **PrimerTextBox**. Snimite je u drugom folderu, recimo **C:\Temp\SoftIng\LecturesCode\Vezba26.**

Otvorite VS .NET i kreirajte novi projekat. Izaberite Visual C# Projects direktorijum unutar prozora Project Types:, i tip projekta Windows Application u okviru prozora Templates: U okviru za tekst Location: promenite putanju u **C:\Temp\SoftIng\LecturesCode\Vezba26**, i ostavite podrazumevani tekst u okviru za tekst Name: **Kontrola ComboBox.**



Da bi smo iskoristili prethodni primer, prvo ćete obrisati iz već postojećeg projekta fajl **Form1.cs** tako što izaberete pomoću desnog tastera miša opciju **Delete** na selektovan fajl.

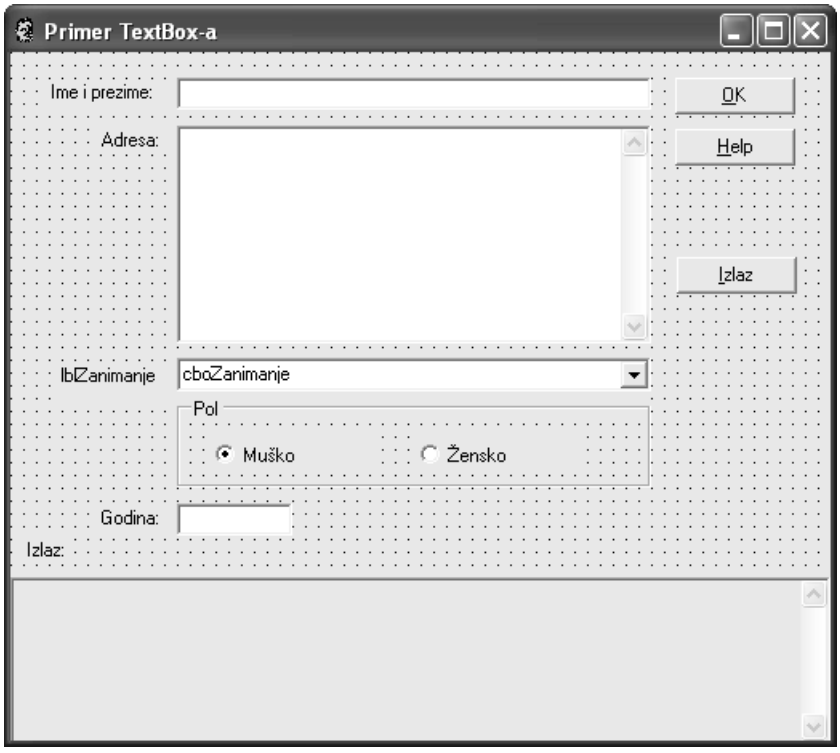
Zatim ćemo dodati fajl sa istim nazivom ali iz prethodnog primera odnosno vežbe 19 koje ste malo pre iskopirali.
Selektujte naziv projekta u prozoru Solution Explorera **KontrolaRadioCheck Button,** izaberete pomoću desnog tastera miša opciju **Add>Add Existing Item...** na selektovan naziv projekta. I birate naziv fajla Form1.cs iz foldera **C:\Temp\SoftIng\LecturesCode\Vezba26\ Kontrola ComboBox.**



I ako ste uradili kako treba, možete otvoriti formu iz prethodnog primera.

Počnimo sa promenom izgleda dijaloga:

Uklonite polje za potvrdu chkProgrammer.
Dodajte oznaku i kombinovani okvir sa imenima, kao što je prikazano na slici:



Podesite svojstva kontrola:

za Label

Osobina (Properties)	Vrednost (Value)
Name	lblZanimanje
Text	Zanimanje:
TextAlign	TopRight
Location	26; 195
Size	72; 23

za ComboBox1

Osobina (Properties)	Vrednost (Value)
Name	cboZanimanje
Location	104; 191
Size	296; 21

Text oznake na **Zanimanje** i izbrišite svojstvo Text kontrole Combo-Box.
Nisu potrebne dalje promene formulara.
Kreirajte u radnom folderu još tekstualnu datoteku **Zanimanje.txt** koja sadrži sledeća dva reda:

Konsultant
Programer

Dodavanje rutina za obradu događaja

Sada možemo menjati kod. Pre nego što počnemo sa pisanjem novog koda, promenićemo kod ratine za obradu događaja btnOK_Click da bi se podudarao sa promenama formulara:

```
329  private void btnOK_Click(object sender, System.EventArgs e)
330  {
331    // Nema provere ispravnosti, pošto nije ni potrebna
332    string izlaz;
333    //Spajamo tekstove cetiri textbox-a
334    izlaz = "Ime i prezime:" + this.txtIme.Text + "\r\n";
335    izlaz += "Adresa:" + this.txtAdresa.Text + "\r\n";
336
337
338    izlaz += "Zanimanje: " + this.cboZanimanje.Text + "\r\n";
339
340
341    izlaz += "Pol: " + (string)(this.rdoZensko.Checked ? "Žensko" : "Muško") + "\r\n";
342    izlaz += "Godina:" + this.txtGodiste.Text;
343    // Ubacujemo novi tekst
344    this.txtIzlaz.Text = izlaz;
345  }
346
```

Umesto polja za potvrdu, sada koristimo kombinovani okvir. Kada je element liste izabran, pojaviće se u tekstualnom delu kontrole ComboBox; zbog ovoga, vrednosti koje nas interesuju uvek će se nalaziti u svojstvu Text kombinovanog okvira.

Sada možemo započeti pisanje novog koda. Pre svega, kreiraćemo metodu koja učitava vrednosti koje već postoje u tekstualnoj datoteci (Zanimanje.txt) i ubacićemo te vrednosti u kombinovani okvir:

```
425   private void LoadZanimanje()
426   {
427
428    try
429    {
430     // Kreiraimo objekat StreamReader. Promenite putanju do mesta
431     // gde vi smeštate datoteku
432     System.IO.StreamReader sr = new System.IO.StreamReader("../../Zanimanje.txt");
433
434     string ulaz;
435     // Citamo dokle god ima redova
436     do
437     {
438      ulaz = sr.ReadLine();
439      // Dodajemo samo ako red sadrzi bar jedan znak
440      if (ulaz != " ")
441         this.cboZanimanje.Items.Add(ulaz);
442     }
443     while (sr.Peek() != -1);
444     // Peek vraca -1 ukoliko je kraj toka
445     // Zatvaramo tok
446     sr.Close();
447    }
448    catch (System.Exception)
449    {
450     MessageBox.Show("Fajl Zanimanje.txt nije naden");
451    }
452   }
```

Čitanje podataka iz ulaznih tokova (pomoću objekta StreamReader) se nećemo ovde baviti, sami pronađite u MSD bibloteci. Dovoljno je red da pomoću ovog objekta otvaramo tekstualnu datoteku **Zanimanje.txt** i čitamo elemente za kombinovani okvir, red po red. Dodajemo svaki red datoteke kombinovanom okviru tako što primenjujemo metodu Add() na kolekciju Items.

Pošto korisnik može uneti novi element u kombinovani okvir, proveravamo slučaj kada je pritisnut taster Enter. Ako tekst iz svojstva Text kombinovanog okvira ne postoji u kolekciji Items, dodajemo novi element toj kolekciji:

```
456   private void cboZanimanje_KeyDown(object sender,
457                 System.Windows.Forms.KeyEventArgs e)
458   {
459    int index = 0;
460    ComboBox cbo = (ComboBox) sender;
461
462    // Radimo nesto samo ako je pritisnut taster ENTER
463    if (e.KeyCode == Keys.Enter)
464    {
465     // FindStringExact trazi string i ne reaguje na mala odnosno velika slova,
466     // jer Programer i programer su isto.
467     // Ako pronadjemo podudarnost, biramo postojeci elemet kolekcije
468     index = cbo.FindStringExact(cbo.Text);
469     if (index < 0 ) // FindStringExact vraca -1 ukoliko nista nije pronadjeno
470      cbo.Items.Add(cbo.Text);
471     else
472      cbo.SelectedIndex = index;
473
474     // Javljamo da smo obradili dogadjaj
475     e.Handled = true;
476    }
477   }
478
```

Metoda FindStringExact () objekta ComboBox traži onaj string koji se u potpunosti podudara sa unetim stringom, ne vodeći računa o malim i velikim slovima. Ovo je savršeno za nas, jer ne želimo da dodamo varijante istog zanimanja u kolekciju.

Ako ne nađemo element u kolekciji Items koji se podudara sa unetim tekstom, dodajemo novi string u kolekciju. Dodavanje novog stringa automatski podešava trenutno izabrani element na onaj koji je upravo unet. Ukoliko pronađemo podudarnost, jednostavno biramo postojeći element kolekcije.

Takođe, potrebno je da se prijavimo na događaj u konstruktoru formulara:

```
62      this.txtGodiste.TextChanged += new
63        System.EventHandler(this.txtBox_TextPromenjen);
64
65      this.cboZanimanje.KeyDown += new
66        System.Windows.Forms.KeyEventHandler(this.cboZanimanje_KeyDown);
67
```

Kada korisnik zatvori dijalog, trebalo bi snimiti elemente kombinovanog okvira. To postižemo sledećom metodom:

```
482   private void SnimiZanimanje()
483   {
484    try
485    {
486     System.IO.StreamWriter sw = new System.IO.StreamWriter("../../Zanimanje.txt");
487     foreach (string item in this.cboZanimanje.Items)
488       sw.WriteLine(item);  // Upisujemo element u datoteku
489     sw.Flush();
490     sw.Close();
491    }
492    catch (System.Exception)
493    {
494     MessageBox.Show("Fajl Zanimanje.txt nije naden !");
495
496   }
497   }
498  }
499 }
500
```

Klasa StreamWriter takođe pogledajte u helpu. Kod za upis teksta u datoteku stavljamo u blok **try.. .catch**, u slučaju da korisnik izbriše ili pomeri tekstualnu datoteku dok je formular još uvek otvoren.

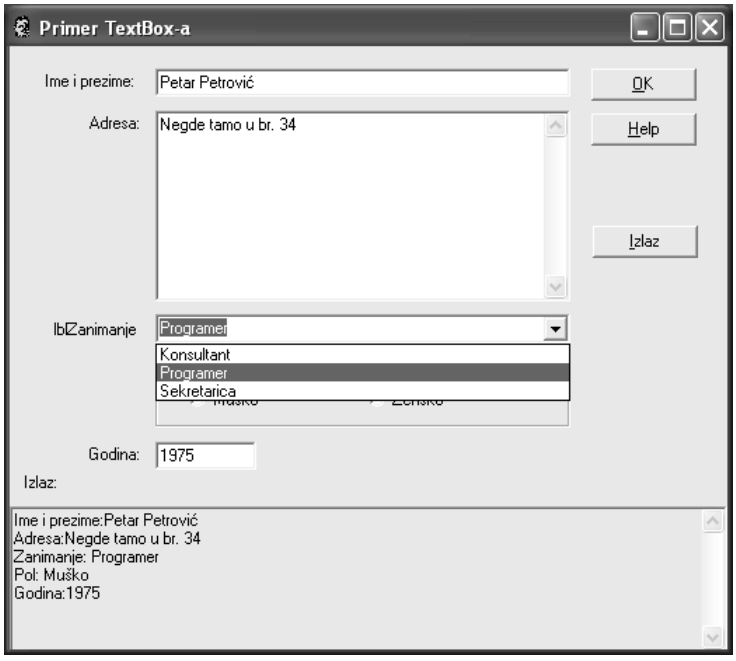
Prolazimo kroz kolekciju Items i upisujemo svaki element u datoteku **Zanimanje.txt**.

Konačno, moramo pozvati metode **LoadOccupation()** i **SaveOccupation()** koje smo upravo kreirali. To radimo u konstruktoru formulara i metodi **Dispose()**, tim redom:

```
34    public fclsMain()
35    {
36      //
37      // Required for Windows Form Designer support
38      //
39      InitializeComponent();
40      this.btnOK.Enabled = false;
41
42      // Podesavamo vrednost Tag na false zbog testiranja
43      // podataka da li su ispravno uneta.
44      this.txtAdresa.Tag = false;
45      this.txtGodiste.Tag = false;
46      this.txtIme.Tag = false;
47
48
49      // Prijavljujemo dogadjaje
50      this.txtIme.Validating += new
51        System.ComponentModel.CancelEventHandler(this.txtBoxPrazan_Validacija);
52      this.txtAdresa.Validating += new
53        System.ComponentModel.CancelEventHandler(this.txtBoxPrazan_Validacija);
          . . . . . .
          . . . . . .
67
68      // Ispunimo kontrolu ComboBox
69      LoadZanimanje();
70    }
71
```

```
75    protected override void Dispose( bool disposing )
76    {
77
78      // Snimamo elemt iz kontrole u ComboBox
79      SnimiZanimanje();
80
81      if( disposing )
82      {
83        if (components != null)
84        {
85          components.Dispose();
86        }
87      }
88      base.Dispose( disposing );
89    }
90
```

Ovim završavamo ComboBox primer. Kada pokrenete program, trebalo bi da dobijete nešto nalik ovome:



Kontrola ListView

Lista iz koje možete birati datoteke, koje želite otvoriti u standardnom Windows dijalogu, zove se kontrola ListView. Sve što možete uraditi vezano za prikaz u standardnom dijalogu (velike ikone, detaljni prikaz itd.) možete uraditi i u kontrolama ListView koje su sastavni deo Visual Studio .NET-a:
Kontrola ListView se obično koristi za prikaz podataka gde korisnik može menjati sol prikazivanja. Podatke koje kontrola sadrži moguće je prikazati različitim ikonama u kolonama i redovima, ili u jednoj koloni. Najčešće korišćena kontrola ListView jeste ona koju vidimo na slici iznad, za navigaciju kroz direktorijume.
Kontrola ListView je najkompleksnija kontrola sa kojom se srećemo u ovom predavanju. Opisati sve njene komponente bilo bi isuviše za ovu knjigu. Međutim, kroz primer ćete videti osnovne informacije o većini najznačajnijih mogućnosti kontrole ListView, kao i detaljne opise svojstava, događaja i metoda. Takođe, pogledaćemo i kontrolu ImageList koja čuva slike korišćene u kontroli ListView.

Svojstva kontrole ListView

Ime osobine	Dostupnost	Opis
Activation	Čitanje/upis	Korišćenjem ove osobine možete kontrolisati kako korisnik aktivira elemente iz kontrole ListView. Ne bi trebalo menjati podrazumevanu vrednost osobine, ukoliko nemate dobar razlog za promenu, zato što se promena podešene vrednosti odnosi na ceo sistem. Moguće vrednosti su: • Standard: ona vrednost koju je korisnik već izabrao za ceo sistem. • OneClick: jedan pritisak mišem aktivira element. • TwoClick: dvostruki pritisak mišem aktivira element.
Alignment	Čitanje/upis	Ovom osobinom regulišete poravnanje elementa u kontroli ListView. Četiri moguće vrednosti su: • Default: element ostaje tamo gde ga je korisnik prevukao i spustio. • Left: elementi su poravnati po levoj ivici kontrole ListView. • Top: elementi su poravnati po gornjoj ivici kontrole ListView. • SnapToGrid: ListView kontrola ima nevidljivu mrežu po kojoj su elementi poravnati.
AllowColumn Reorder	Čitanje/upis	Ako podesite ovu osobinu na true, dozvoljavate korisniku da menja redosled kolona. Ako to uradite, morate imati rutine koje će valjano ubacivati elemente u listu, bez obzira na to da li je redosled kolona promenjen ili nije.
AutoArrange	Čitanje/upis	Ako podesite ovu osobinu na true, element će zauzeti položaj prema vrednosti svojstva Alignment. Ako je, na primer, svojstvo Alignment podešeno na Left, kada korisnik prevuče element do centra kontrole, onda se element automatski pomera na levu ivicu kontrole. Podešavanje ove osobine ima efekta samo ako je svojstvo View podešeno na Largelcon ili Smalllcon.
CheckBoxes	Čitanje/upis	Ako podesite ovu osobinu na true, svaki element u listi imaće polje za potvrdu levo od sebe. Podešavanje ove osobine ima efekta samo ako je svojstvo View podešeno na Details ili List.
Checkedlndices Checkedlterns	Samo čitanje	Ova dva svojstva daju pristup kolekcijama koje sadrže indekse i elements koji su potvrđeni.
Columns	Samo čitanje	Kontrola ListView može imati kolone. Ovo svojstvo obezbeđuje pristup kolekciji kolona, kroz koju možete dodavati ili uklanjati kolone.
Focusedltem	Samo čitanje	Ovo svojstvo sadrži element koji ima fokus. Ako ništa nije izabrano, vrednost je null.

Ime osobine	Dostupnost	Opis
FullRowSelect	Čitanje/upis	Ako je ovo svojstvo podešeno na true, kada se pritisne na neki element, onda će ceo red u kom se element nalazi biti naglašen. Ako je podešeno na false, samo će element biti naglašen.
GridLines	Čitanje/upis	Podešavanjem ove osobine na true pokazaće se linije mreže između redova i kolona. Podešavanje ove osobine ima efekta samo ako je svojstvo view podešeno na Details.
HeaderStyle	Čitanje/upis	Možete kontrolisati kako će biti prikazano zaglavlje kolone. Postoje tri stila: • Clickable: zaglavlje kolone se ponaša kao dugme. • NonClickable: zaglavlje kolone ne reaguje na pritisak mišem. • None: zaglavlja kolona nisu prikazana.
HoverSelection	Čitanje/upis	Kada je ovo svojstvo podešeno na true, korisnik može izabrati element postavljanjem miša na element.
Items	Samo čitanje	Kolekcija elemenata u lisa.
LabelEdit	Čitanje/upis	Kada je vrednost ovog svojstva podešena na true, korisnik može uređivati sadržaj prve kolone u prikazu Details.
LabelWrap	Čitanje/upis	Kada je ovo svojstvo podešeno na true, oznake će se protezati na koliko god je redova potrebno za prikazivanje celog teksta.
LargelmageList	Čitanje/upis	Ovo svojstvo sadrži kontrolu ImageList u kojoj se nalaze velike slike. Ove slike se mogu koristiti kada je svojstvo View podešeno na Largelcon.
MultiSelect	Čitanje/upis	Podesite ovu osobinu na true, ako hoćete da dozvolite korisniku da izabere vise elemenata odjednom.
Scrollable	Čitanje/upis	Ako podesite ovu osobinu na true, pokazaće se traka za pomeranje sadržaja.
Selectedlndices	Samo čitanje	Sadrži indekse izabranih elemenata.
Selectedltems	Samo čitanje	Sadrži izabrane elemente.
SmallImageList	Čitanje/upis	Ovo svojstvo sadrži kontrolu ImageList u kojoj se nalaze slike. Ove slike se mogu koristiti kada je svojstvo View podešeno na Smalllcon.
Sorting	Čitanje/upis	Kontrola ListView može sortirati elemente koje sadrži. Postoje tri različita načina: • Ascending: po rastućem redosledu. • Descending: po opadajućem redosledu. • None: nema sortiranja.
StatelmageList	Čitanje/upis	Kontrola ImageList sadrži maske za slike koje se koriste za prekrivanje slika iz kolekcija LargelmageList i SmalllmageList, u cilju prikazivanja trenutnog stanja.
Topitern	Samo čitanje	Vraća element koji se nalazi na vrhu liste.
View	Čitanje/upis	Kontrola može pokazati elemente u četiri različita režima: • Largelcon: svi elementi su prikazani kao velike ikone (32 x 32) sa oznakama. • Smalllcon: svi elementi su prikazani kao male ikone (16 x 16) sa oznakama. • List: prikazana je samo jedna kolona koja može da sadrži ikonu i oznaku. • Details: bilo koji broj kolona može biti prikazan. Samo prva kolona može da sadrži ikonu.

Metode klase ListView

Za kompleksnu kontrolu kakva je ListView, postoji iznenađujuće mali broj metoda. Sve su opisane u sledećoj tabeli:

Ime	Opis
BeginUpdate	Pozivanjem ove metode stopirate osvežavanje (ažuriranje) slike, sve dok se ne pozove metoda EndUpdate. Ovo može biti korisno, pogotovu ako ubacujete veliki broj elemenata odjednom; sprečava se treptanje prikaza i značajno se povećava brzina.
Clear	Kompletno uklanja sve elemente i kolone kontrole ListView.
EndUpdate	Ovu metodu pozivate posle metode BeginUpdate. Kada je pozovete, svi elementi će biti iscrtani u kontroli ListView.
EnsureVisible	Kada pozovete ovu metodu, lista će se pomeriti do onog elementa čiji je indeks prosleđen metodi.
GetItemAt	Vraća element u listi koji se nalazi na poziciji x, y.

Događaji kontrole ListView

Sledeća tabela opisuje događaje koje možete pozivati za kontrolu ListView:

Ime	Opis
AfterLabelEdit	Ovaj događaj nastaje posle uređivanja oznake.
BeforeLabelEdit	Ovaj događaj nastaje pre nego što korisnik započne uređivanje oznake.
ColumnClick	Ovaj događaj nastaje kada je kolona pritisnuta.
ItemActivate	Dešava se kada se element aktivira.

Klasa ListViewItem

Element liste je uvek objekat klase Listviewltem. Klasa ListViewItem sadrži informacije, kao što su tekst i indeks ikone, koje treba prikazati. Takođe, sadrži kolekciju koja se zove SubItems. Ova kolekcija sadrži objekte klase ListViewSubItems. Ovi podelementi su prikazani ako je kontrola ListView u režimu Details. Svaki od podelemenata predstavlja kolonu. Osnovna razlika između elementa i podelementa jeste ta što podelement ne može prikazati ikonu. Objekte ListViewItem dodajete kontroli ListView kroz kolekciju Items. Objekte ListViewSubItems dodajete kontroli ListViewItem kroz kolekciju SubItems.

Klasa ColumnHeader

Za prikazivanje zaglavlja kolona, dodajete objekte klase ColumnHeader kontroli List-View, kroz kolekciju Columns. Klasa ColumnHeader obezbeđuje zaglavlje za kolone koje će se prikazati kada je ListView u režimu Details.

Kontrola ImageList

Kontrola ImageList obezbeđuje kolekciju koja se koristi za smeštanje slika koje mogu biti korišćene u drugim kontrolama. U liste slika mogu se stavljati slike bilo koje veličine, s tim da se u okviru jedne kontrole mogu naći samo slike iste veličine. U slučaju kontrole ListView potrebne su dve kontrole ImageList za smeštanje malih i velikih slika. ImageList je prva kontrola s kojom se srećemo u ovom predavanju koja ne prikazuje svoj sadržaj u vremenu izvršavanja programa. Kada dovučete ovu kontrolu na formular za vreme dizajniranja, ona ne postaje deo samog formulara već se nalazi u bloku ispod formulara, koji smešta sve slične objekte. Ovo zanimljivo svojstvo ne dopušta kontrolama koje nisu deo grafičkog interfejsa da zatrpavaju dizajner formulara. Kontrolom ImageList se rukuje kao i svakom drugom kontrolom, s tim da je ne možete

pomerati.
Slike se mogu dodavati kontroli ImageList i u vreme dizajniranja i u vreme izvršavanja programa. Ako unapred znate koje slike želite prikazati, možete pritisnuti na dugme desno od svojstva Images. Prikazaće se dijalog u kojem možete doći do željenih slika. Ako se odlučite za dodavanje slika u toku izvršavanja programa, to možete postići kroz kolekciju Images.

Primer kontrole ListView

Napravićemo dijalog sa kontrolom ListView i dve kontrole ImageList. ListView će prikazati datoteke i direktorijume smeštene na vašem disku. Zbog jednostavnosti, nećemo izvlačiti tačne ikone datoteka i direktorijuma, već ćemo koristiti standardne ikone za direktorijume i datoteke. Otvaranje direktorijuma omogućiće vam da pretražujete stablo direktorijuma, a dugme Back da se vratite unazad. Četiri radio dugmeta koristićemo za promenu prikaza liste u toku izvršavanja programa. Ako se dva puta pritisne neka datoteka, pokušaćemo da je otvorimo (izvršimo) i uvek, počinjemo kreiranjem grafičkog interfejsa:

Vežba br. 27.

Napravite novi projekat konzolne aplikacije birajući File | New | Project... iz menija:

Izaberite Visual C# Projects direktorijum unutar prozora Project Types:, i tip projekta Windows Application u okviru prozora Templates: U okviru za tekst Location: promenite putanju u **C:\Temp\SoftIng\LecturesCode\Vezba27** (ovaj će direktorijum biti automatski napravljen ukoliko već ne postoji), i ostavite podrazumevani tekst u okviru za tekst Name: **Kontrola ListView**.

Otvorite paletu Properties i za kontrolu ***forme*** podesite sledeće osobine:

Osobina (Properties)	Vrednost (Value)
Name	fclsMain
Text	Kontrola ListView
StartPosition	CenterScreen
FormBorderStyle	FixedSingle
Size	558; 325

Dodajte kontrolu Labele, i podesite sledeće osobine:

Osobina (Properties)	Vrednost (Value)
Name	lblTrenutnaPutanja
Text	
Location	16; 8
Size	528; 16

Dodajte kontrolu dugmeta, i za kontrolu *Button* podesite sledeće osobine:

Osobina (Properties)	Vrednost (Value)
Name	btnNazad
Text	Nazad
Location	240; 256
Size	75; 23

Dodajte jednu kontrolu grupnog okvira ili **GroupBox** i u nju četiri kontrole radio dugmeta **Radio Button** grupnom okviru i podesite njihove osobine u prozoru Properties, redom.

za GroupBox1

Osobina (Properties)	Vrednost (Value)
Name	groPregledMod
Text	Pregled mod
Location	424; 32
Size	120; 128

za četiri kontrole **RadioGroup**

Osobina (Properties)	Vrednost (Value) za kontrolu			
	RadioGroup1	**RadioGroup2**	**RadioGroup3**	**RadioGroup4**
Name	rdoLargeIcon	rdoSmallIcon	rdoList	rdoDetails
Text	Velike Ikone	Male Ikone	List Pregled	Detalj Pregled
Checked	false	false	true	false
Location	8; 24	8; 48	8; 72	8; 96
Size	96; 16	104; 16	104; 16	104; 16

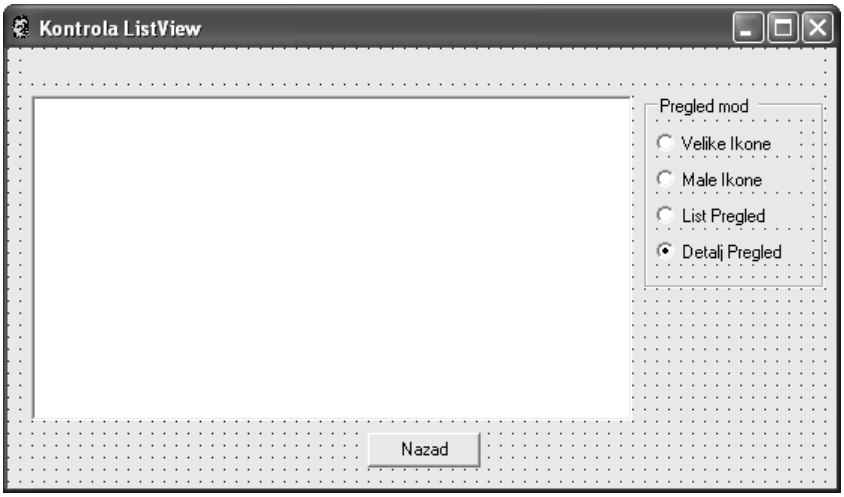
Dodajte dve nevidljive kontrole **ImageList** na formu tako što ćete dva puta pritisnuti ikonu kontrole u okviru sa alatima (moraćete da ga pomerite nadole da biste pronašli ikonu). Promenite svojstva u prozoru Properties za **ImageList** kontrole: Vrednost (Value) za kontrolu

Osobina (Properties)		
	ImageList1	**ImageList2**
Name	ilMaleIkone	ilVelikeIkone
ImageSize	16; 16	32; 32
Images	<Oznaka diska>:\Visual Studio Files\Graphics\Icons\Win95**16X16 Folder.ico** <Oznaka diska>:\Visual Studio Files\Graphics\Icons\Win95**16X16 Files.ico**	

Dodajte kontrolu ListView, i za kontrolu *ListView* podesite sledeće osobine:

Osobina (Properties)	Vrednost (Value)
Name	lwFajloviFolderi
View	List
LargeImageList	ilVelikeIkone
SmallImageList	ilMaleIkone
MultiSelect	true
Location	16; 32
Size	558; 325

Dobijate formular kao na slici:



Dodavanje rutina za obradu događaja

Ovim završavamo sa grafičkim interfejsom i prelazimo na kod. Pre svega, treba nam polje u koje će se smeštati direktorijumi kroz koje smo već prošli, tako da se možemo vratiti na njih kada se pritisne dugme **Nazad**. Čuvaćemo apsolutne putanje do direktorijuma, i za taj posao koristimo StringCollection:

```
18  public class Form1 : System.Windows.Forms.Form
19  {
20   // Polje koje ce sadrzati prethodne foldere
21   private System.Collections.Specialized.StringCollection folderCol;
22
23
```

Nismo napravili zaglavlje kolone; zato ćemo to sada uraditi. Pravimo ih u metodi KreirajHeaderListView():

```
210   private void KreirajHeaderListView()
211   {
212    ColumnHeader colHead;
213
214    // Prvo zaglavlje
215    colHead = new ColumnHeader();
216    colHead.Text = "Naziv fajla";
217    this.lwFajloviFolderi.Columns.Add(colHead); // Ubacimo zaglavlje
218
219    // Drugo zaglavlje
220    colHead = new ColumnHeader();
221    colHead.Text = "Veličina fajla";
222    this.lwFajloviFolderi.Columns.Add(colHead); // Ubacimo zaglavlje
223
224    // Trece zaglavlje
225    colHead = new ColumnHeader();
226    colHead.Text = "Poslednji pristup";
227    this.lwFajloviFolderi.Columns.Add(colHead); // Ubacimo zaglavlje
228   }
```

Počinjemo deklarisanjem promenljive colHead, koja se koristi za kreiranje tri zaglavlja. Za svako od ova tom zaglavlja kreiramo novu promenljivu, i dodeljujemo joj svojstvo Text pre nego što je dodamo u kolekciju Columns kontrole ListView.

Ostaje još inicijalizacija formulara sadržajem koji se prikazuje na početku, a to je popunjavanje kontrole ListView datotekama i direktorijumima sa diska. Ovo radimo u sledećem metodu:

```
230   private void FilujListView(string root)
231   {
232    try
233    {
234     // Dve lokalne promenljive za kreiranje elemenata za ubacivanje
235     ListViewItem lvi;
236     ListViewItem.ListViewSubItem lvsi;
237
238     // Ako nema korenog foldera, nista ne moze ubaciti
239     if (root.CompareTo("") == 0)
240      return;
241
242     // Uzimamo informacije o korenom folderu.
243     System.IO.DirectoryInfo dir = new System.IO.DirectoryInfo(root);
244
245     // Citamo fajlove i foldere iz korenog foldera.
246     DirectoryInfo[] dirs = dir.GetDirectories(); // Folderi
247     FileInfo[] files = dir.GetFiles(); // Fajlovi
248
249     // Brisemo ListView. Zovemo metodu Clear kolekcije Items, a ne same
250     // kontrole ListView. Metoda Clear kontrole ListView brise sve
251     // ukljucujuci i zaglavlja kolona, a mi hocemo da izbrisemo samo elemente
252     // iz prikaza.
253     this.lwFajloviFolderi.Items.Clear();
254
255     // Podesimo oznaku na trenutnu putanju
256     this.lblCurrentPath.Text = root;
257
258     //Stopiramo oznaku na trenutnu putanju.
259     this.lwFajloviFolderi.BeginUpdate();
260
261     // Prodjimo kroz foldere korenog foldera i ubacimo ih
262     foreach (System.IO.DirectoryInfo di in dirs)
263     {
264      // Kreiramo glavni ListViewItem.
265      lvi = new ListViewItem();
266      lvi.Text = di.Name; // Ime Foldera
267      lvi.ImageIndex = 0; // Indeks ikone foldera je 0
268      lvi.Tag = di.FullName; // Podesimo tag na putanju foldera
269
270      // Kreiramo dva podelementa.
271      lvsi = new ListViewItem.ListViewSubItem();
272      lvsi.Text = ""; // Folder nema velicinu i zato je ova kolona prazna
273      lvi.SubItems.Add(lvsi); // Dodajemo podelement u ListViewItem
274
275      lvsi = new ListViewItem.ListViewSubItem();
276      lvsi.Text = di.LastAccessTime.ToString(); // Poslednja kolona kojoj je pristup
277      lvi.SubItems.Add(lvsi); // Dodajemo podelement u ListViewItem
278
279      // Dodajemo ListViewItem kolekciji Items kontrole ListView
280      this.lwFajloviFolderi.Items.Add(lvi);
281     }
```

```
283         // Prolazimo kroz sve fajlove korenog foldera
284         foreach (System.IO.FileInfo fi in files)
285         {
286           // Kreiramo glavni ListViewItem.
287           lvi = new ListViewItem();
288           lvi.Text = fi.Name; // Ime Fajla
289           lvi.ImageIndex = 1; // Ikona kojom prikazujemo folder ima indeks 1
290           lvi.Tag = fi.FullName; // Podesimo tag na putanju foldera
291
292           // Kreiramo dva podelementa.
293           lvsi = new ListViewItem.ListViewSubItem();
294           lvsi.Text = fi.Length.ToString(); // Velicina fajla
295           lvi.SubItems.Add(lvsi); // Dodajemo podelemente kolekciji SubItems.
296
297           lvsi = new ListViewItem.ListViewSubItem();
298           lvsi.Text = fi.LastAccessTime.ToString(); // Poslednja kolona kojoj pristupamo
299           lvi.SubItems.Add(lvsi); // Dodajemo podelemente kolekciji SubItems.
300
301           // Dodajemo element kolekciji Items kontroli ListView
302           this.lwFajloviFolderi.Items.Add(lvi);
303         }
304
305         // Omogucujemo prokaz novih elemenata. Sada se prikazuju novih elemenata.
306         this.lwFajloviFolderi.EndUpdate();
307       }
308       catch (System.Exception err)
309       {
310         MessageBox.Show("Error: " + err.Message);
311       }
312     }
```

Pre prvog od dva bloka **foreach**, pozivamo metod BeginUpdate() za kontrolu ListView. Setite se da metoda **BeginUpdate()** stopira osvežavanje (ažuriranje) slike sve dok se ne pozove metoda **EndUpdate().** Da nismo dodali ovu metodu, ispunjavanje liste bi bilo znatno sporije i slika bi treptala kako se elementi dodaju. Neposredno posle drugog bloka foreach pozivamo metodu EndUpdate(), koja prikazuje sve elemente kontrole ListView.

Posebno nas interesuju dva bloka foreach. Počinjemo tako što kreiramo novi objekat klase **ListViewItem** i podešavamo svojstvo Text na ime datoteke ili direktorijuma koji ćemo ubaciti. Svojstvo ImageIndex objekta ListViewItem odnosi se na indeks elementa u jednoj od kontrola ImageList. Zbog ovoga, važno je da ikone imaju iste indekse u obe kontrole ImageList. Koristimo svojstvo Tag za smeštanje putanje do datoteka i direktorijuma, koju koristimo kada korisnik dva puta pritisne element.

Zatim, kreiramo dva podelementa. Njima je dodeljen tekst za prikazivanje, posle čega dodajemo podelemente kolekciji SubItems objekta ListViewItem.

Konačno, objekat ListViewItem dodat je kolekciji Items kontrole ListView. Kontrola ListView je dovoljno pametna da ignoriše podelemente ukoliko režim prikaza liste nije Details. Zato, dodajemo podelemente bez obzira na trenutni režim.

Ostaje jedino da prikažemo koreni direktorijum pozivanjem dve funkcije u konstruktoru tormulara. U isto vreme stvaramo instancu klase StringCollection pod imenom folderCol i inicijalizujemo je korenim direktorijumom:

```
36    public Form1()
37    {
38      //
39      // Required for Windows Form Designer support
40      //
41      InitializeComponent();
42
43      // Inicijalizujemo ListView i kolekciju foldera
44      folderCol = new System.Collections.Specialized.StringCollection();
45      KreirajHeaderListView();
46      FilujListView(@"C:\");
47      folderCol.Add(@"C:\");
48
```

Da bi korisnik bio u stanju da,dva puta pritisne element u kontroli ListView i tako pretražuje direktorijume, moramo da prijavimo događaj ItemActivate. To radimo u konstruktoru:

```
50
51      this.lwFajloviFolderi.ItemActivate += new
52              System.EventHandler(this.lwFajloviFolderi_ItemActivate);
53    }
```

Odgovarajuća ratina za obradu ovog događaja izgleda ovako:

```
321   private void lwFajloviFolderi_ItemActivate(object sender, System.EventArgs e)
322   {
323     // Konvertujemo parametar sender u tip ListView  i citamo svojstvo tag za
324     // prvi izabrani element.
325     System.Windows.Forms.ListView lw = (System.Windows.Forms.ListView)sender;
326     string filename = lw.SelectedItems[0].Tag.ToString();
327
328     if (lw.SelectedItems[0].ImageIndex != 0)
329     {
330       try
331       {
332         // Pokusavamo da pokrenemo fajl
333         System.Diagnostics.Process.Start(filename);
334       }
335       catch
336       {
337         // Ako pokusaj nije uspeo, napustamo metodu.
338         return;
339       }
340     }
341     else
342     {
343       // Ubacujemo elemente.
344       FilujListView(filename);
345       folderCol.Add(filename);
346     }
347   }
348
```

Svojstvo **tag** izabranog elementa sadrži punu putanju do datoteka i direktorijuma na koje se dva puta pritisne. Znamo da su slike sa indeksom 0 direktorijumi; prema tome, možemo utvrditi da li je element datoteka ili direktorijum ako pogledamo u njegov indeks. Ako je datoteka, pokušaćemo da ga otvorimo. Ako je direktorijum, pozivamo metodu **PaintListView** sa putanjom novog direktorijuma. Zatim, dodajemo novi direktorijum kolekciji **folderCol**.

Pre nego što predemo na radio dugmad, završićemo mogućnosti pretraživanja dodajući rutinu za obradu pritiska dugmeta Back. Dva puta pritisnite dugme Back i dodajte sledeći kod:

```
349     private void btnBack_Click(object sender, System.EventArgs e)
350     {
351       if (folderCol.Count > 1)
352       {
353         FilujListView(folderCol[folderCol.Count-2].ToString());
354         folderCol.RemoveAt(folderCol.Count-1);
355       }
356       else
357       {
358         FilujListView(folderCol[0].ToString());
359       }
360     }
```

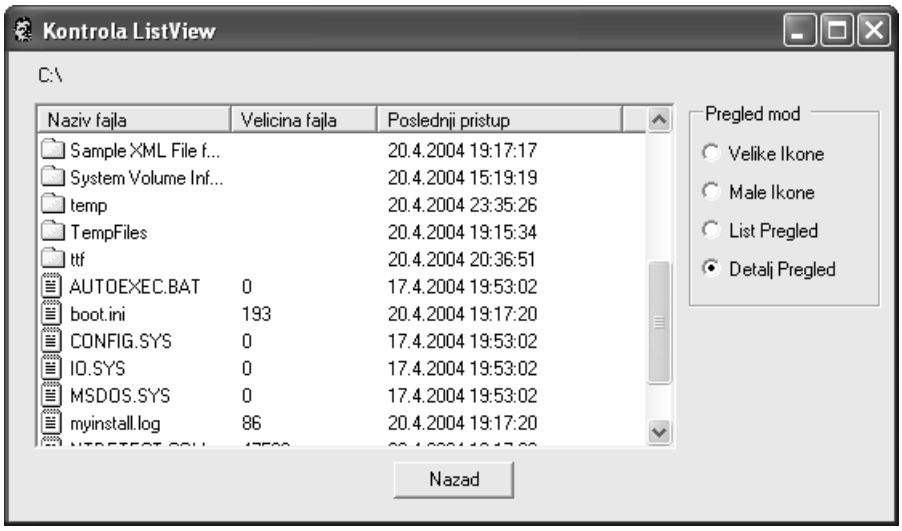
Ako postoji vise od jednog elementa u kolekciji folderCol, znad da nismo u korenom direktorijumu, pa pozivamo metod PaintListView sa putanjom do prethodnog direktorijuma. Poslednji element u kolekciji folderCol jeste direktorijum u kom se trenutno nalazimo; zbog toga treba uzeti pretposlednji direktorijum. Zatim, uklanjamo poslednji element kolekcije i pomeramo se na pretposlednji. Ukoliko postoji samo jedan element u kolekciji, jednostavno pozivamo metodu PainListView sa tim elementom.

Ostaje dodavanje koda za četiri radio dugmeta, da bismo mogli menjati tip prikaza liste. Dva puta pritisnite svako radio dugme i dodajte sledeći kod:

```
362     private void rdoLargeIcon_CheckedChanged(object sender, System.EventArgs e)
363     {
364       RadioButton rdb = (RadioButton)sender;
365       if (rdb.Checked)
366         this.lwFajloviFolderi.View = View.LargeIcon;
367     }
368
369     private void rdoList_CheckedChanged(object sender, System.EventArgs e)
370     {
371       RadioButton rdb = (RadioButton)sender;
372       if (rdb.Checked)
373         this.lwFajloviFolderi.View = View.List;
374     }
375
376     private void rdoSmallIcon_CheckedChanged(object sender, System.EventArgs e)
377     {
378       RadioButton rdb = (RadioButton)sender;
379       if (rdb.Checked)
380         this.lwFajloviFolderi.View = View.SmallIcon;
381     }
382
383     private void rdoDetails_CheckedChanged(object sender, System.EventArgs e)
384     {
385       RadioButton rdb = (RadioButton)sender;
386       if (rdb.Checked)
387         this.lwFajloviFolderi.View = View.Details;
388     }
389   }
390 }
391
```

Proveravamo svojstvo Checked za svako radio dugme. Ako je svojstvo podešeno na true, podešavamo svojstvo View kontrole ListView shodno tome.

Ovim završavamo ListView primer. Kada pokrenete program, trebalo bi da dobijete nešto nalik ovome:



Kontrola StatusBar

Statusna linija se najčešće koristi za prikazivanje uputstva za izabrani element ili za informaciju o akciji koja se trenutno izvodi. Uobičajeno, kontrola StatusBar se postavlja u dnu ekrana, kao što je to slučaj sa MS Office aplikacijama, ali može se staviti i na neko drugo mesto. Visual Studio.NET statusna linija se koristi za prikazivanje teksta. Takođe, možete dodavati panele za prikaz teksta, kao i sopstvene ratine za iscrtavanje sadržaja panela:

Slika iznad prikazuje statusnu liniju MS Worda. Paneli statusne linije, tj. sekcije, prikazuju se kao ulegnute.

Svojstva kontrole StatusBar

Kao što smo već rekli, možete podesiti svojstvo Text statusne linije za prikazivanje nekog teksta, ali, takođe je moguće praviti panele u iste svrhe:

Ime osobine	Dostupnost	Opis
BackgroundImage	Čitanje/upis	Moguće je dodeliti sliku statusnoj liniji, koja će se iscrtati u pozadini.
Panels	Čitanje/upis	Ovo je kolekcija panela u statusnoj liniji. Koristite ovu kolekciju za dodavanje i uklanjanje panela.
ShowPanels	Samo čitanje	Ako hoćete da prikažete panele, ovo svojstvo mora biti podešeno na true.
Text	Čitanje/upis	Kada ne koristite panele, ovo svojstvo sadrži tekst koji će se prikazati na statusnoj liniji.

Događaji kontrole StatusBar

Nema mnogo novih događaja za statusnu liniju, ali ako crtate panel ručno, događaj DrawItem je od velike važnosti:

Ime	Opis
DrawItem	Dešava se kada panel sa Owner Draw stilom treba ponovo iscrtati. Morate prijaviti ovaj događaj ako želite sami da crtate sadržaj panela.
PanelClick	Dešava se kada se izabere panel.

Klasa StatusBarPanel

Svaki panel u statusnoj linije jeste objekat klase StatusBarPanel. Ova klasa sadrži informacije o svim panelima kolekcije Panels. Može se podešavati tekst, položaj teksta u panelu, ikone ili stil panela.

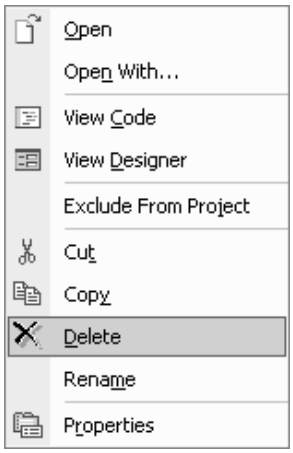
Ako želite sami da crtate panel, morate podesiti panel-ovo svojstvo Style na OwnerDraw i obraditi događaj DrawItem kontrole StatusBar.

Primer kontrole StatusBar

Vežba br. 28.

Izmenićemo postojeći primer ListView kako bismo prikazali upotrebu kontrole StatusBar.

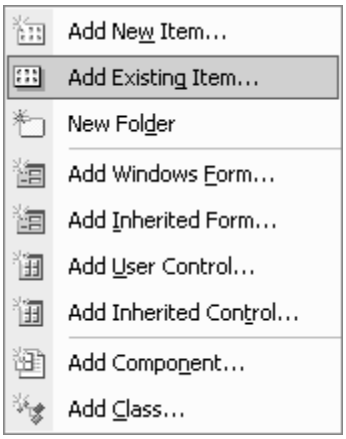
Stari projekat odnosno vežbu pod brojem 27. **Kontrola ListView**. Snimite je u drugom folderu, recimo **C:\Temp\SoftIng\LecturesCode\Vezba28.**



Otvorite VS .NET i kreirajte novi projekat. Izaberite Visual C# Projects direktorijum unutar prozora Project Types:, i tip projekta Windows Application u okviru prozora Templates: U okviru za tekst Location: promenite putanju u **C:\Temp\SoftIng\LecturesCode\Vezba28**, i ostavite podrazumevani tekst u okviru za tekst Name: **Kontrola StatusBar.**

Da bi smo iskoristili prethodni primer, prvo ćete obrisati iz već postojećeg projekta fajl **Form1.cs** tako što izaberete pomoću desnog tastera miša opciju **Delete** na selektovan fajl.

Zatim ćemo dodati fajl sa istim nazivom ali iz prethodnog primera odnosno vežbe 27 koje ste malo pre iskopirali.



Selektujte naziv projekta u prozoru Solution Explorera **Kontrola StatusBar**, izaberete pomoću desnog tastera miša opciju **Add>Add Existing Item...** na selektovan naziv projekta. I birate naziv fajla Form1.cs iz foldera **C:\Temp\SoftIng\LecturesCode\Vezba28\ Kontrola StatusBar.**

I ako ste uradili kako treba, možete otvoriti formu iz prethodnog primera.

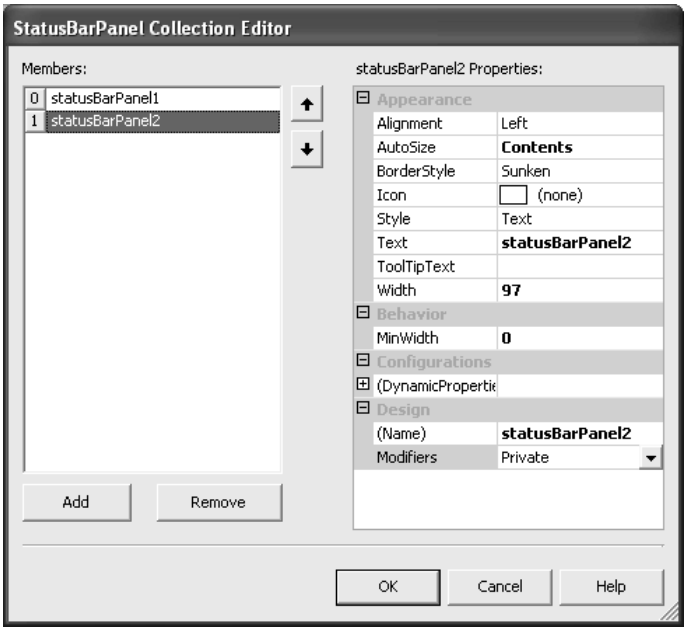
Počnimo sa promenom izgleda dijaloga:

Izbacićemo oznaku odnosno labelu, koju smo koristili za prikaz postojećeg direktorijuma, i umesto toga ubaciti te informacije na panel statusne linije. Takođe, dodaćemo još jedan panel za prikaz trenutnog režima prikazivanja liste:

Uklonite oznaku **lblTrenutnaPutanja**.

Dva puta pritisnite kontrolu StatusBar u okviru sa alatima da biste je ubacili u formular. Kontrola će se automatski pripojiti donjoj ivici formulara i podesite njene osobine u prozoru properties.

Osobina (Properties)	Vrednost (Value)
Name	sbInfo
Text	izbrišite
Panels	StatusBarPanel1 AutoSize: Spring; Ovo znači da će panel deliti prostor statusne linije sa ostalim panelima. StatusBarPanel2 AutoSize: Contents; Ovo znači da će panel menjati veličinu u zavisnosti od veličine teksta koji sadrži. MinWidth: 0
ShowPanels	true



Dodavanje rutina za obradu događaja

Završili smo sa grafičkim interfejsom i prelazimo na kod. Počinjemo sa podešavanjem trenutne putanje u metodi **FilujListView**. Izbrišite liniju koja podešava tekst u oznaci i umesto nje ubacite sledeći kod:

```
286
287         this.sbInfo.Panels[0].Text = root;
288         this.sbInfo.Panels[1].Text="List Pregled";
```

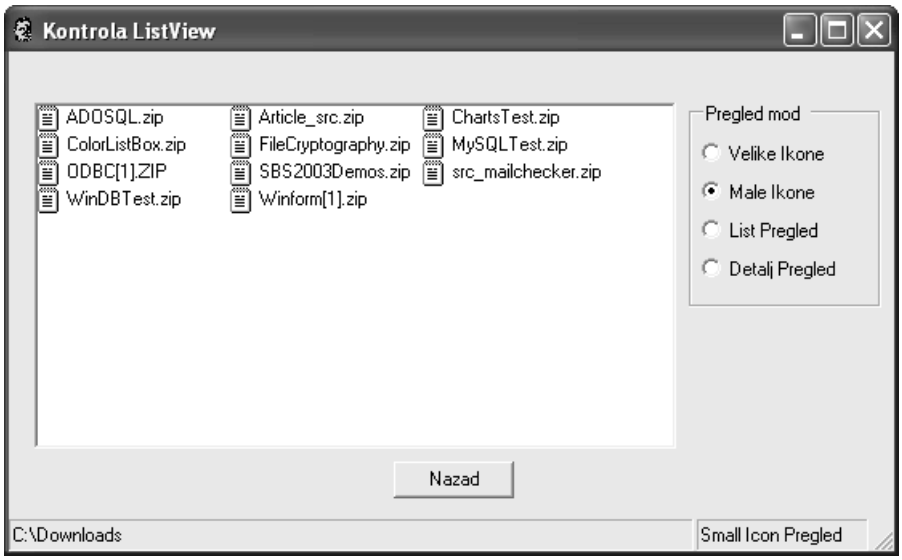
Prvi panel ima indeks **0**, zato podešavamo svojstvo Text kao što smo podesili svojstvo Text oznake. Drugi red koda podešava prikaz drugog panela, zato ima indeks **1**. Inicijalizujemo tekst drugog panela na List Pregled.

Na kraju, podešavamo rutine za obradu događaja CheckedChanged za četiri kontrole RadioButton, tako da prikažu tekst u drugom panelu:

```
392     private void rdoLargeIcon_CheckedChanged(object sender, System.EventArgs e)
393     {
394       RadioButton rdb = (RadioButton)sender;
395       if (rdb.Checked)
396       {
397         this.lwFajloviFolderi.View = View.LargeIcon;
398         this.sbInfo.Panels[1].Text="Large Icon Pregled";
399       }
400     }
401
402     private void rdoList_CheckedChanged(object sender, System.EventArgs e)
403     {
404       RadioButton rdb = (RadioButton)sender;
405       if (rdb.Checked)
406       {
407         this.lwFajloviFolderi.View = View.List;
408         this.sbInfo.Panels[1].Text="List Pregled";
409       }
410     }
411
412     private void rdoSmallIcon_CheckedChanged(object sender, System.EventArgs e)
413     {
414       RadioButton rdb = (RadioButton)sender;
415       if (rdb.Checked)
416       {
417         this.lwFajloviFolderi.View = View.SmallIcon;
418         this.sbInfo.Panels[1].Text="Small Icon Pregled";
419       }
420     }
421
```

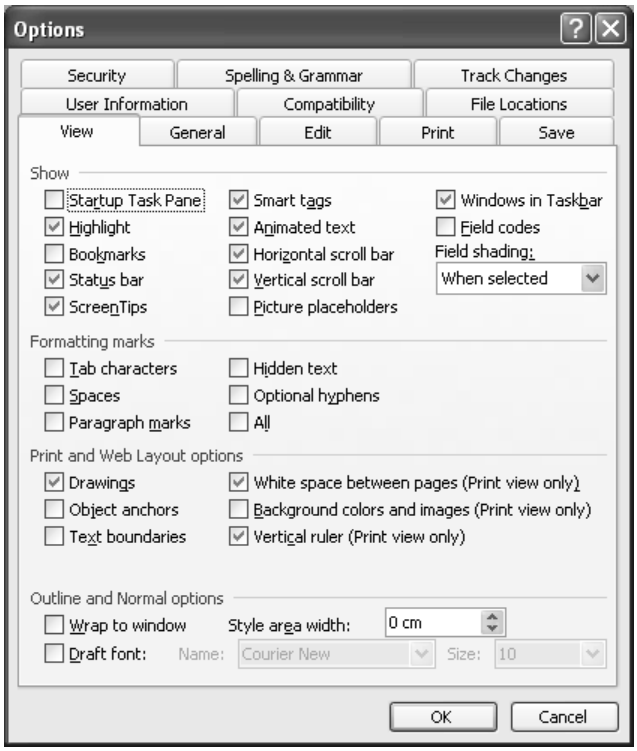
```
422     private void rdoDetails_CheckedChanged(object sender, System.EventArgs e)
423     {
424       RadioButton rdb = (RadioButton)sender;
425       if (rdb.Checked)
426       {
427         this.lwFajloviFolderi.View = View.Details;
428         this.sbInfo.Panels[1].Text="Details Pregled";
429       }
430     }
431   }
432 }
433
```

Ovim završavamo StatusBar primer. Ako pokrenete program, trebalo bi da dobijete nešto nalik ovome:



Kontrola TabControl

Kontrola TabControl obezbeđuje lak način organizovanja dijaloga u logičke jedinice kojima se može pristupiti kroz jezičke kartica na vrhu kontrole. Kontrola TabControl sadrži kontrole TabPages koje su slične grupnim okvirima, s tim što su nešto kompleksnije:



Slika na prethodnoj strani pokazuje dijalog Options MS Worda 2000, u podrazumevanoj konfiguraciji. Primetite dva reda koja sadrže jezičke kartica u vrhu kontrole. Pritiskom na bilo koji od njih prikazaće se drugačija selekcija kontrola u ostatku dijaloga. Ovo je dobar primer kako grupisati informacije koje su u nekoj vezi jedna s drugom, tako olakšavajući korisniku da pronade željenu informaciju.

Korišćenje kontrole TabControl je veoma lako. Jednostavno dodate jezičke za kartice koje želite prikazati u kolekciju TabPages, a zatim ubacujete ostale kontrole u odgovarajuće strane.
Za vreme izvršavanja programa, možete se kretati kroz kartice koristeći svojstva kontrole.

Svojstva kontrole TabControl

Osobine kontrole TabControl se uglavnom koriste za podešavanje izgleda kontejnera u kome se nalaze kontrole TabPages, tj. za izgled kartica:

Ime osobine	Dostupnost	Opis
Alignment	Čitanje/upis	Kontroliše gde se na kontroli pojavljuju jezičci kartica. Podrazumeva se da su na vrhu.
Appearance	Čitanje/upis	Kontroliše kako su prikazani jezičci kartica. Mogu biri kao dugme ili ravni.
HotTrack	Čitanje/upis	Ako je ovo svojstvo podešeno na true, izgled jezička kartice se menja kako pokazivač miša prelazi preko njega.
Multiline	Čitanje/upis	Ako je ovo svojstvo podešeno na true, moguće je imati nekoliko redova sa jezičcima kartica.
RowCount	Samo čitanje	Sadrži broj redova sa jezičcima kartica koji su trenutno prikazani.
Selectedlndex	Čitanje/upis	Vraća ili podešava indeks trenutno izabranog jezička kartice.
TabCount	Čitanje/upis	Sadrži ukupan broj kartica.
TabPages	Čitanje/upis	Ovo je kolekcija objekata TabPages. Koristite ovu kolekciju za dodavanje ili uklanjanje kartica.

Rad sa kontrolom TabControl

Kontrola TabControl ponaša se malo drugačije u odnosu na ostale kontrole koje smo do sada videli. Kada dovučete kontrolu na formular, videćete sive četvorouglove, koji ne izgledaju kao kontrole koje vidimo na slici ekrana iznad. Takođe, videćete ispod panela Properties, dva dugmeta koja izgledaju kao hiperveze sa natpisima Add Tab i Remove Tab.

Ako pritisnete dugme Add Tab, ubacićete novu karticu u kontrolu, a kontrola će početi da liči na sebe. Očigledno, izbacićete karticu iz kontrole ako pritisnete dugme Remove Tab.

Ova procedura je obezbedena radi lakšeg korišćenja kontrole. Ako, sa druge strane, želite da promenite ponašanje ili stil kartice, trebalo bi da koristite dijalog TabPages. Do ovog dijaloga dolazite kada u panelu sa svojstvima izaberete TabPages.
Svojstvo Tab Pages je i kolekcija koju možete koristiti za pristup pojedinačnim stranicama kontrole. Kreiraćemo primer koji demonstrira osnovne karakteristike kontrole TabControl.

Vežba br. 29.

Napravite novi projekat konzolne aplikacije birajući File | New | Project... iz menija:

Izaberite Visual C# Projects direktorijum unutar prozora Project Types:, i tip projekta Windows Application u okviru prozora Templates: U okviru za tekst Location: promenite putanju u **C:\Temp\SoftIng\LecturesCode\Vezba29** (ovaj će direktorijum biti automatski napravljen ukoliko već ne postoji), i ostavite podrazumevani tekst u okviru za tekst Name: **Kontrola TabControl**.

Otvorite paletu Properties i za kontrolu ***forme*** podesite sledeće osobine:

Osobina (Properties)	Vrednost (Value)
Name	fclsMain
Text	Kontrola TabControl
StartPosition	CenterScreen
FormBorderStyle	FixedSingle
Size	298; 305

Dodajte kontrolu TabControl, i podesite sledeće osobine:

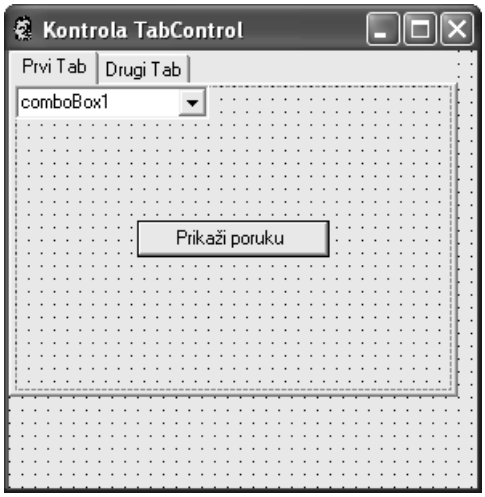
Osobina (Properties)	Vrednost (Value)
Name	tbcKontrola
TabPages	TabControl1 Text: Prvi Tab TabControl2 Text: Drugi Tab
Location	0; 0
Size	280; 216

Na prvoj TabPages odnosno TabControl1 postavite jednu kontrolu ComboBox i jednu kontrolu dugmeta Button i podesite joj osobine:

Osobina (Properties)	Vrednost (Value)
Name	btnMessages
Text	Prikaži poruku
Location	76; 84
Size	120; 23

Na drugoj TabPages odnosno TabControl2 postavite jednu kontrolu TextBox i podesite joj osobine:

Osobina (Properties)	Vrednost (Value)
Name	txtMessage
Text	obrišite
Location	56; 56
Size	152; 20



Dodavanje rutine za obradu događaja

Sada smo spremni da pristupamo kontrolama. Ako pokrenete kod u ovom obliku, videćete kartice prikazane kako treba. Ostaje jedino da demonstriramo rad kontrole - kada korisnik pritisne na Show Message dugme na jednoj kartici, prikazaće se tekst koji je unet u tekstualni okvir na drugoj kartici. Prvo, dodajemo rutinu za obradu događaja Click. Dva puta pritisnite dugme na prvoj kartici i dodajte sledeći kod:

```
152
153     private void btnMessages_Click(object sender, System.EventArgs e)
154     {
155       MessageBox.Show(this.txtMessage.Text);
156     }
157   }
158 }
159
```

Kontroli na jednoj stranici pristupamo kao i svakoj drugoj kontroli. Dolazimo do svojstva Text kontrole TextBox i prikazujemo je u okviru za poruke.
Već smo videli da je moguće izabrati samo jedno radio dugme na formularu (ukoliko ih ne stavite u odvojene grupne okvire). Kartice kontrole TabControl ponašaju se isto kao i grupni okviri, tako da je moguće imati različite grupe radio dugmadi na različitim karticama ove kontrole, bez korišćenja grupnih okvira.

Još jednu stvar morate da znate kada radite sa kontrolama TabControl - koja kartica je trenutno prikazana. Dve osobine mogu pomoći: SelectedTab i Selectedlndex. Kao što imena govore, svojstvo SelectedTab vraća objekat TabPage, ili null ukoliko nijedna kartica nije izabrana. Svojstvo Selectedlndex vraća indeks kartice, ili -1 ukoliko nijedna kartica nije izabrana.

Sažetak

U ovom predavanju videli smo najčešće korišćene kontrole za izradu Windows aplikacija i kako se one mogu koristiti za pravljenje jednostavnih ali moćnih korisničkih interfejsa. Opisali smo osobine i događaje tih kontrola i dali primere njihove upotrebe.

U ovom predavanju obradili smo sledeće kontrole:

- Label
- Button
- RadioButton
- CheckBox
- ComboBox
- ListBox
- ListView
- GroupBox
- RichTextBox
- StatusBar
- ImageList
- TabControl

U sledećem predavanju pogledaćemo kompleksnije kontrole kao što su meniji i palete alata. Koristićemo ih za razvoj Windows aplikacija koje podržavaju rad sa vise dokumenata {Multi-Document Interface - MDI). Takođe, pokazaćemo kako se kreiraju korisničke kontrole koje kombinuju funkcionalnost jednostavnih kontrola, a koje smo obradili u ovom predavanju.