UNIVERZITET METROPOLITAN BEOGRAD

C# PROGRAMSKI JEZIK

CILJ

Ovladavanje teorijom i praktičnim primenama programskog jezika C#. Proučavaće se pažljivo izabrana serija primera primene C#, sa kompletnim objašnjenjima. Sve važne osobine i karakteristike ovog programskog jezika biće prodiskutovane i ilustrovane kao i instaliranje i upotreba integrisanog softverskog okruženja za C#. Ovo znanje može da se koristi za pripremu nastave kao i u radu sa učenicima.

Plan realizacije kursa

Vremenski plan realizacije kursa

Trajanje kursa izraženo u satima i danima:

24 sata, odnosno 3 radna dana, najviše po 8 sati dnevno, pri čemu je kompletna realizacija online (online 24 sata) i to:

15 sati za proučavanje lekcije,
30 minuta za test,
2 sata foruma,
6.5 sati za vežbanje zadataka i komunikaciju sa realizatorom.

Plan realizacije po časovima i temama

- Instalacija Visual Studio 2010 IDE.

čas	Tema	Nastavna pitanja
1.	Instalacija Visual Studio 2010 IDE	1. Preduslovi 2. Lokacija instalacionih datoteka 3. Postupak instalacije 4. PostInstalaciona podešavanja
2.	C# i .NET Framework	1. Platforma .NET 2. .NET Framework 3. Prevođenje i MSIL 4. Jezik C# - nastanak 5. Konzolna aplikacija „Hello World“

3.	Osnove jezika C#	1. Tipovi podataka

		2. Promenljive i konstante 3. Naredbe 4. Operatori
4.	Konzolne aplikacije	1. Postupak kreiranja konzolnih aplikacija 2. Konzolna aplikacija – sabiranje stringova 3. Konzolna aplikacija – operacije sa celim brojevima 4. Konzolna aplikacija – unos celog broja
5.	Klase i objekti	1. Klase i objekti 2. Kreiranje klase 3. Kreiranje objekata 4. Korišćenje ključne reči this
6.	Klase i objekti	1. Metode 2. Konstruktori i destruktori 3. Modifikatori pristupa 4. Enkapsulacija
7.	Logičke strukture	1. Naredbe grananja 2. Petlje
8.	Nasleđivanje, interfejsi i apstraktne klase	1. Nasleđivanje 2. Interfejsi 3. Apstraktne klase
9.	Polimorfizam i overriding	1. Polimorfizam 2. Overriding
10.	Strukture, enumeratori i referenciranje i konverzija	1. Strukture podataka 2. Enumeratori 3. Referenciranje i konverzija
11.	Nizovi	1. Pojam nizova 2. Deklaracija nizova 3. Konstrukcija nizova 4. Inicijalizacija nizova 5. Dvodimenzionalni i višedimenzionalni nizovi 6. Rad sa elementima niza
12.	Delegati, događaji i kolekcije	1. Delegati 2. Događaji 3. Kolekcije
13.	Izuzeci	1. Izuzeci 2. Često korišćene klase izuzetaka

14.	Windows kontrole	1. Osobine Windows kontrola 2. Izbor i postavljanje kontrole na formu 3. Selekcija kontrole, podešavanje pozicije i velicine 4. Zajednička svojstva Windows kontrola 5. Postavljanje kontrole u fokus
15.	Windows kontrole – primeri korišćenja	1. Button 2. TextBox 3. Label 4. GroupBox 5. Panel 6. PictureBox 7. OpenFile Dialog
		8. Radio button 9. RadioBox 10. CheckBox 11. ListBox 12. MessageBox

Nastavni čas 1: Instalacija Visual Studio 2010 IDE

Za realizaciju gotovo svih nastavnih sadržaja iz ovog predmeta koristiće se Microsoft Visual Studio 2010, te studenti koji nemaju na svojim radnim stanicama instalirano razvojno okruženje, neka izvrše download sa metropolitan sajta i instaliraju razvojno okruženje.

Preduslovi:

Softverski zahtevi:

Visual Studio 2010 može biti instaliran na sledećim operativnim sistemima:

- Windows XP (x86) - Service Pack 3 – sve edicije osim Starter edicije
- Windows Vista (x86 & x64) - Service Pack 2 - sve edicije osim Starter edicije
- Windows 7 (x86 & x64)
- Windows Server 2003 (x86 & x64) - Service Pack 2
- Windows Server 2003 R2 (x86 & x64)
- Windows Server 2008 (x86 & x64) - Service Pack 2
- Windows Server 2008 R2 (x64)

Podržane arhitekture:

- 32-Bit (x86)

- 64-Bit (x64)

Hardverski zahtevi:

- 1.6GHz ili brži procesor
- 1 GB (32 Bit) ili 2 GB (64 Bit) RAM
- Ukoliko se razvojno okruženje instalira na virtuelnu mašinu potrebno je dodati najmanje 512 MB virtuelnoj mašini
- 3GB prostora na hard disku
- 5400 RPM hard disk
- DirectX 9 video card @1024 x 768 ili većoj rezoluciji
- DVD-ROM Drive ukoliko se ne instalira sa download-ovanog fajla.

Lokacija instalacionih datoteka

Za potrebe realizacije svih predmeta koji se bave Microsoft tehnologijama, sa studentskog portala univerziteta Metropolitan u okviru MSDNAA svaki student može izvršiti download potrebnog software-a.

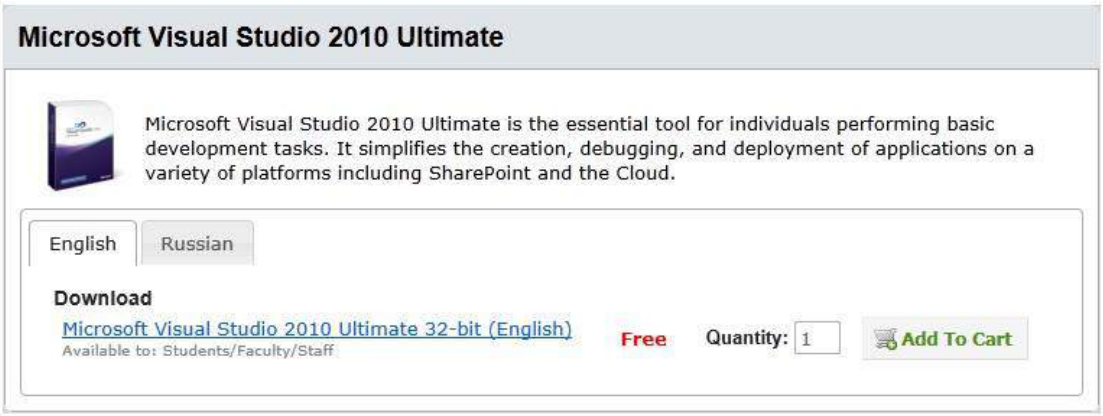
Za realizaciju ovog predmeta potreban je MS Visual Studio 2010 i on se takođe može downloadovati sa portala, preko linka Microsoft software (bilo da pristupate sa sajta univerziteta Metropolitan ili sa sajta Fakulteta Informacionih tehnologija).

Napomena:

Za download softvera je potrebna registracija, te ukoliko to već niste učinili u nekom od prethodnih perioda, učinite što pre i preuzmite alat.

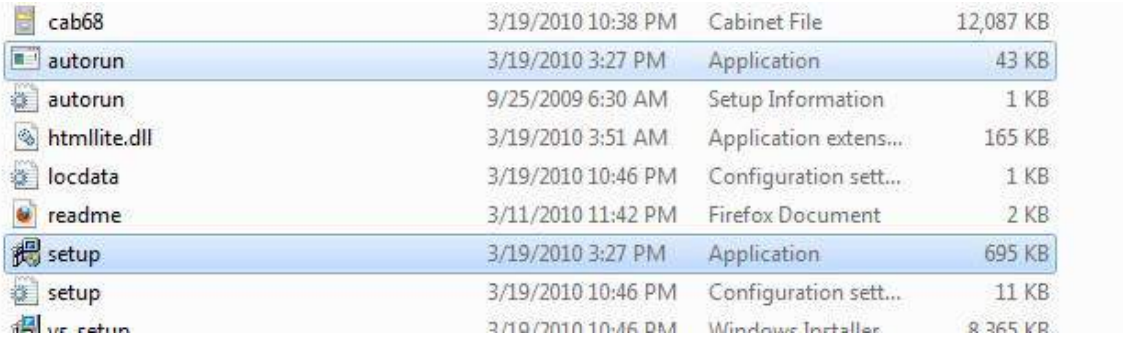
Postupak instalacije

Po logovanju na sistem, vrši se postupak downloada datoteke. Source je u iso formatu (en_visual_studio_2010_ultimate_x86_dvd_509116.iso), te je potrebno iso datoteku pripremiti za instalaciju.

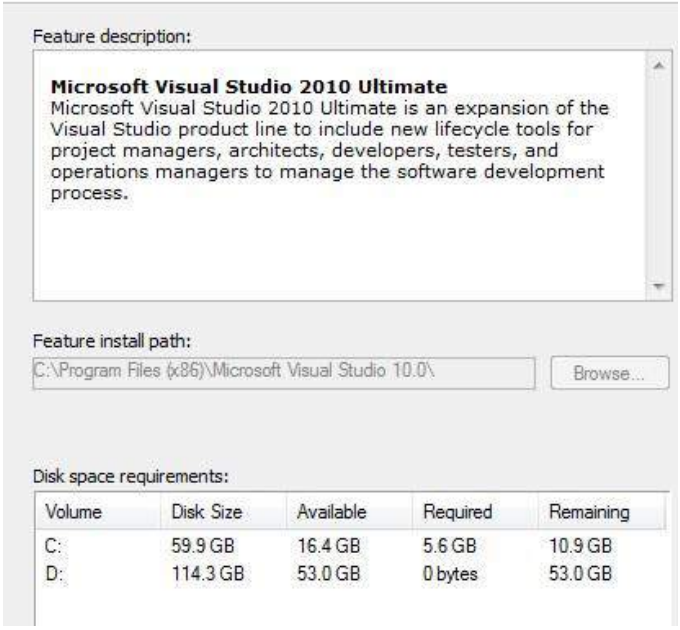


Najjednostavniji način pripreme (raspakivanja) je smeštanje iso datoteke na optički medijum, čime se ona prilikom smeštanja raspakuje u egzekucioni format, a ukoliko ne želite da smeštate datoteku na optički medijum možete izvršiti instalaciju putem nekog virtuelnog diska ili sličnog programa koji rukuje iso fajlovima.

Po raspakivanju iso fajla, potrebno je pokrenuti setup (duplim klikom na setup.exe ili autorun.exe) i instalacija se odvija automatski preko instalation wizard-a.



U delu instalacione pripreme potrebno je odabrati odgovarajuću lokaciju (ili ostaviti default podešavanje) i prepustiti čarobnjaku da izvrši raspakivanje i instaliranje potrebnih fajlova.



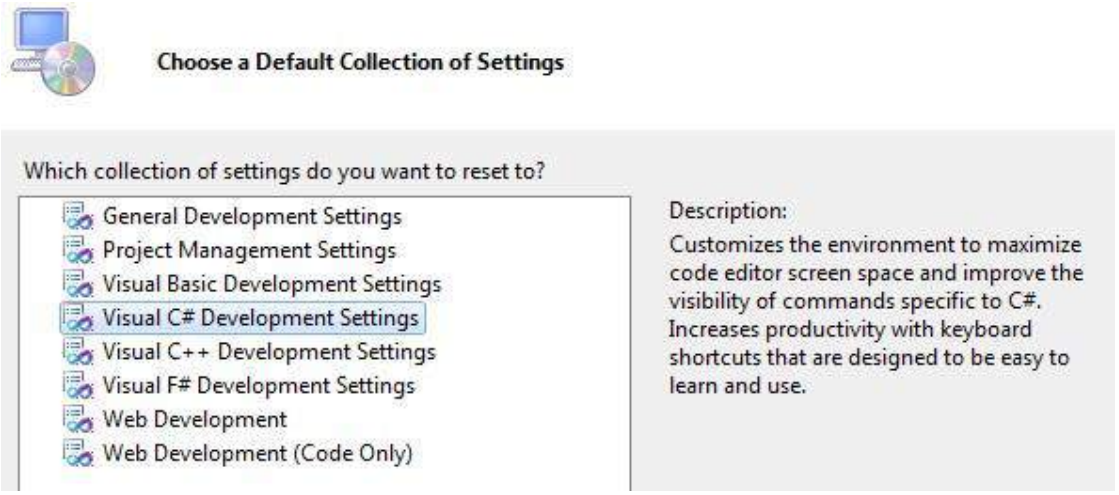
Nakon iybora lokacije instalacioni čarobnjak će Vas voditi do samog završetka instalacije.

Prilikom instalacije izvršiće se izmena postojeće platforme, biće dodata podrška za .NET 4.0 platformu, update raznih alata (u slučaju da ste ranije imali neki instalirani alat, primera radi SilverLight, prilikom instalacije biće Vam ponuđena opcija zamene itd).

Po završenoj instalaciji, preporučljivo je izvršiti restart računara.

PostInstalaciona podešavanja

Po završenoj instalaciji i prvom pokretanju VisualStudio2010 će Vas pitati da postavite default podešavanja. Za potrebe ovog predmeta potrebno je odabrati Visual C# Development Settings.



Napomena:

U svakom trenutku je moguće promeniti default development setings preko tools -> import and export settings.

Nastavni čas br. 2: C# i .NET Framework

Platforma .NET

.NET je razvojna platforma s novim interfejsom za programiranje aplikacija (engl. application programming interface , API); nasledio je funkcionalnost i mogućnosti okruženja za programiranje klasičnih operativnih sistema Windows, ali i usvojio brojne, različite tehnologije koje Microsoft razvija od kraja devedesetih godina prošlog veka. To obuhvata rad s komponentama COM+ i XML-om, objektno orijentisan dizajn, podršku za nove protokole za Web servise kakvi su SOAP, WSDL i UDDI, i orijentisanost ka Internetu; sve je to integrisano u okviru DNA arhitekture (Distrubuted interNet Applications).

Microsoft je mnogo resursa stavio u službu razvoja platforme .NET i srodnih tehnologija. Opseg tehnologija koje obuhvata .NET je veliki. Platformu čine sledeće tri grupe tehnologija:

Skup jezika – uključujući C# i VB, skup alatki za razvoj programa (izmeođu ostalog, Visual Studio .NET), opsežna biblioteka klasa za pravljenje Web servisa i aplikacija za Web i Windows, i zajedničko izvršno okruženje (engl. Common Language Runtime , CLR) za izvršavaçe koda objekata napravàenih u .NET Frameworku

Dve generacije servera .NET Enterprise Server: postojeći, i oni koji će postati dostupni u naredne dve do tri godine

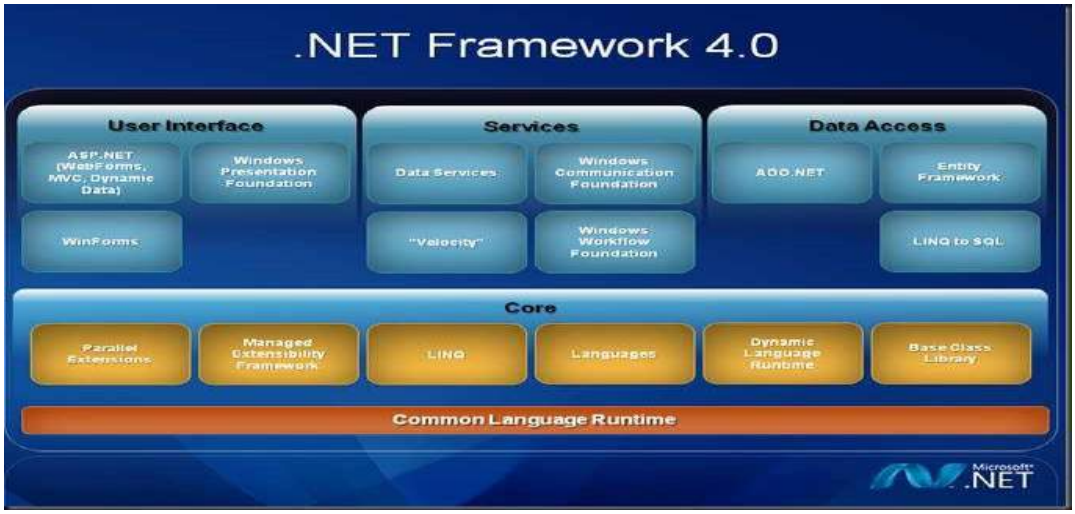
.NET podrška koja nije samo za PC računare, već i za nove ureođaje – od mobilnih telefona, do platformi za kompjuterske igrice

.NET Framework

Reč je o programskom okviru, operativnom sistemu unutar operativnog sistema, koji su programeri kompanije Microsoft razvili kako bi nezavisni programeri mogli lakše da razvijaju svoje programe. .NET Framework je paket kodova neophodan programerima aplikacija za uobičajene probleme u programiranju, koji uključuje i virtuelnu mašinu koja upravlja izvršavanjem programa pisanih za .NET Framework. Ta virtuelna mašina, nazvana Common Language Runtime (CLR), je softversko okruženje u kojem se izvršavaju programi pisani za .NET Framework. Sa CLR, programeri ne moraju da razmišljaju o kapacitetima procesora koji će izvršavati njihove programe. Pored toga CLR obezbeđuje i druge važne elemente kao što su bezbednost aplikacije, upravljanje memorijom i rukovanje izuzecima.

.NET Framework uključuje i ogromnu biblioteku kodova i podataka. On podržava nekoliko programskih jezika što obezbeđuje jezičku interoperabilnost, tako da svaki programski jezik može koristiti kod pisan drugim programskim jezikom. Biblioteka klasa .NET Framework-a obezbeđuje korisnički interfejs, pristup podacima, kriptografiju, razvoj veb aplikacija, numeričke algoritme i mrežnu komunikaciju. Programeri kombinuju biblioteke klasa sa sopstvenim kodom za kreiranje svojih programa.

Mnoge aplikacije koje su dela nezavisnih programera kao preduslov za instalaciju i rad zahtevaju da neka verzija .NET Framework-a bude instalirana na računaru. Šta više, u većini slučajeva aplikacije će od vas zahtevati instalaciju određene verzije NET.Framework, instaliranu na računaru. Preporuka je da izbegavate instaliranje zahtevane verzije, i umesto toga instalirate najnoviju verziju .NET-a, pod predpostavkom da je vaša verzija Windows-a podržava. Većina .NET paketa je kompatibilna sa starijim verzijama Framework-a, tako da ukoliko aplikacija za instalaciju zahteva verziju 2.0 .NET-a, sve što joj je potrebno nalazi se i u najnovijoj verziji - .NET Framework 4. Naravno, prethodno morate da imate instalirane sve relevantne zakrpe za Windows kako bi .NET Framework bio kompatibilan sa vašim sistemom.



Struktura .NET Framework-a je data na slici iznad.

Ono što može biti problem prilikom instalacije .NET Framework-a jeste potreban prostor na hard disku. Verzija 4.0 za 32-bitne Windows sisteme zahteva 850 MB slobodnog prostora na primarnom disku, dok je za 64-bitne verzije potrebno 2 GB slobodnog prostora. Windows neće od vas zahtevati da obezbedite prostor za instalaciju ukoliko imate dovoljno slobodnog prostora na drugoj particiji. Ukoliko nemate takav prostor, instalacija neće biti odobrena i najpre ćete morati da oslobodite prostor na disku kako biste ponovo otpočeli proces instalacije.

NAPOMENA: .NET Framework se instalira zajedno sa VisualStudio IDE, ali je ovde obrađen .NET Framework koji je moguće instalirati I odvojeno od MSVS IDE.

Prevođenje i MSIL

U okruženju .NET, programi se ne prevode u izvršne datoteke, već u programske sklopove koji sadrže instrukcije na Microsoftovom međuejeziku (engl. Microsoft Intermediate Language , MSIL), koje CLR prevodi na mašinski kôd i izvršava. MSIL (skraćeno IL) datoteke koje pravi C# skoro su identične IL datotekama nastalim u drugim .NET jezicima; za platformu je nevažno na kom je jeziku program napisan.

Ključna odlika okruženja CLR jeste to da je zajedničko svim .NET jezicima – na isti način podržava pisanje programa na C#-u kao i na jeziku VB.NET.

Kôd napisan na C#-u prevodi se na IL u fazi prevođenja (engl. Build) projekta.

Međuejezički kôd se čuva u datoteci na disku. Kada pokrenete program, međuejezički kôd se ponovo prevodi pomoću pravovremenog prevodioca (engl. Just In Time compiler, JIT compiler). Ovaj proces se na engleskom često naziva JITing. Kao rezultat, dobija se mašinski kôd koji izvršava procesor računara.

Standardni JIT prevodioci rade po zahtevu (engl. on demand). Kada se pozove metoda, JIT prevodilac analizira međeujezički kôd i pravi vrlo efikasan mašinski kôd koji se veoma brzo izvršava. Dok se izvršava program, JIT prevodilac prevodi samo po potrebi, a prevedeni kôd se čuva u memoriji da bi se mogao ponovo iskoristiti. Tokom izvršavanja, .NET programi postaju sve brži, jer se koristi već preveden kôd.

Suština CLS-a je da svi .NET jezici proizvode vrlo sličan međujezički kôd. Zato svaki jezik može pristupati objektima napisanim na drugom jeziku i izvoditi nove objekte od njih. Dakle, moguće je napraviti osnovnu klasu na jeziku VB.NET i na C#-u iz nje izvesti novu klasu.

Jezik C# - nastanak

C# je jednostavan objektno-orijentisan programski jezik opšte namene. Razvio ga je Microsoft tim koji je vodio Andres Hejlsberg. Poslednja verzija C# je 4.0 koja je završena 12. Aprila 2010. god. Prva verzija (C# 1.0) se pojavila 2001.godine, pa su se ubrzo pojavljivale nove verzije ovog programskog jezika. C# predstavlja naslednika C i C++ jezika, dobio je ime sharp po inspiraciji muzičke notacije i znači da se napisana nota izvodi za pola koraka više. C# je naprednija verzija C++ (C++++). Fajlovi pisani u ovom jeziku imaju ekstenziju cs.

Bitno je napomenuti da C# programski jezik sintaksno nije složen (ima oko 80 rezervisanih reči), ali je vrlo izražajan u delu gde je potrebno rešiti bilo kakav problem softveskog development procesa. Podržava strukturirano, objektno orijentisano programiranje zasnovano na komponentama, u potpunosti podržava sve principe OOP I SOA, podržava interfejse nasleživanje, strukture, delegate, komponentno orijentisane elemente (svojstva, događaji, deklaracije).

Pored mogućnosti koje se tiču same implementacije programskih jezika, bitno je napomenuti da C# podržava i sledeće:

- Direktan pristup memoriji pomoću pokazivača u stilu C++-a
- Rezervisane reči za izdvajanje nesigurnih operacija
- Upozoravaçe skupljača smeća okruženja CLR da ne uništava objekte na koje pokazuju pokazivači dok se ti objekti ne oslobode, itd.

Pored svega gore navedenog, C# podržava još mnogo elemenata programiranja, ali to u ovom delu nećemo razmatrati, jer nam nije cilj pobrojavanje elemenata koje C# može da podrži nego shvatanje njegove moći.

Konzolna aplikacija „Hello World“

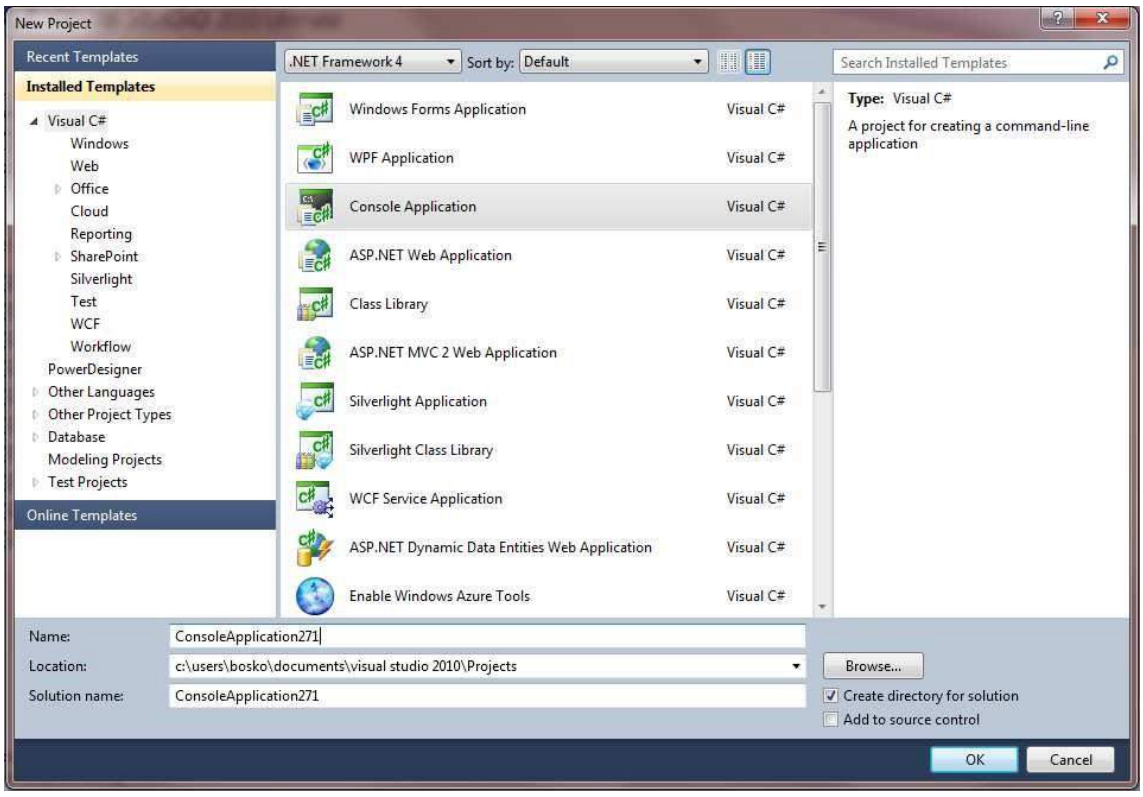
Po tradiciji izučavanja svih programskih jezika, prilikom realizacije prve aplikacije koristi se hello world konzolna aplikacija. Ni ovoga puta nećemo odstupiti od toga.

Da biste kreirali aplikaciju Hello World, potrebno je da kreirate konzolnu aplikaciju. Rad sa konzolnim aplikacijama biće obrađen kasnije, ali u ovom delu će biti prikazan način kreiranja Hello World aplikacije.

Postupak je sledeći: Na slici ispod je prikazana startna strana, na kojoj se mogu odabrati ponuđene mogućnosti za formiranje novog projekta izborom opcije New Project ili otvaranje već postojećeg projekta izborom opcije Open Project.

Pri kreiranju konzolne aplikacije izabere se opcija sa padajućeg menija File→New Project.

Zatim se u okviru panela Templates izabere Console Application. Potom se dobija sledeće:



Nakon unosa imena, lokacije i solution-a, okruženje Vam automatski kreira kod koji je dovoljan za pokretanje jedne konzolne aplikacije (listing koda Vam je dat ispod).

```
using System;
```

```
using
System.Collections.Generic;
using System.Linq; using
System.Text;

namespace ConsoleApplication271
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

Da bismo razumeli šta nam je okruženje generisalo u kodu, daćemo i kratak opis koda:

namespace

predstavlja .NET način da se kod i njegov sadržaj jedinstveno identifikuju. Takođe se koristi za kategorizaciju elemenata u .NET Framework. namespace je deklarisan za kod aplikacije koja se u ovom slučaju naziva ConsoleApplication271.

using System;

predstavlja korišćenje System namespace (odnosi se na biblioteku klasa .NET Framework, čiju biblioteku koristi C#). Za veće projekte može se kreirati sopstveni namespaces.

Analogno sa ovim objašnjenjem su i objašnjenja za Collections.Generic, Linq I Text.

classPrimer{

definicija klase Primer. Početak klase počinje sa otvorenom vitičastom zagradom {, a završava se sa zatvorenom vitičastom zagradom }.

static voidMain()

počinje glavni metod, gde se izvršava program. Ovaj metod počinje sa static, što znači da je statički (ne može da se instancira, odnosno ne može od njega da se napravi objekat) i void - ne vraća vrednost.

Kako bismo napravili naš prvi program potrebno je da uradimo malo editovanje koda I da naš program nateramo da izvrši ispis na ekran. To ćemo uraditi sledećom naredbom:

```
Console.WriteLine("Hello World.");
```

Objašnjenje: WriteLine() je funkcija koji kao rezultat vraća string koji mu je prosleđen. Console- podržava konzolni I/O. U kombinaciji se kompajleru govori da je WriteLine() članica Console klase. Izrazi u C# se zvršavaju sa tačka-zarezom (;).

Kada bismo pokrenuli naš program, verovatno bismo videli da se podigla konzola, a zatim odmah i zatvorila. Razlog ovoga je to što smo rekli da se izvrši ispis na ekran, to je računar I uradio I odmah nakon toga je nastavio dalje. Kako nema drugih instrukcija, a radi se o main metodi, vraća int vrednost (u default slučaju 0) i zatvara konzolu. Da bismo predupredili ovakvo ponašanje, potrebno je da zahtevamo da se desi neka akcija od strane korisnika. Najjednostavnije je da tražimo unos nekog karaktera. To ćemo postići na sledeći način:

```
Console.ReadKey();
```

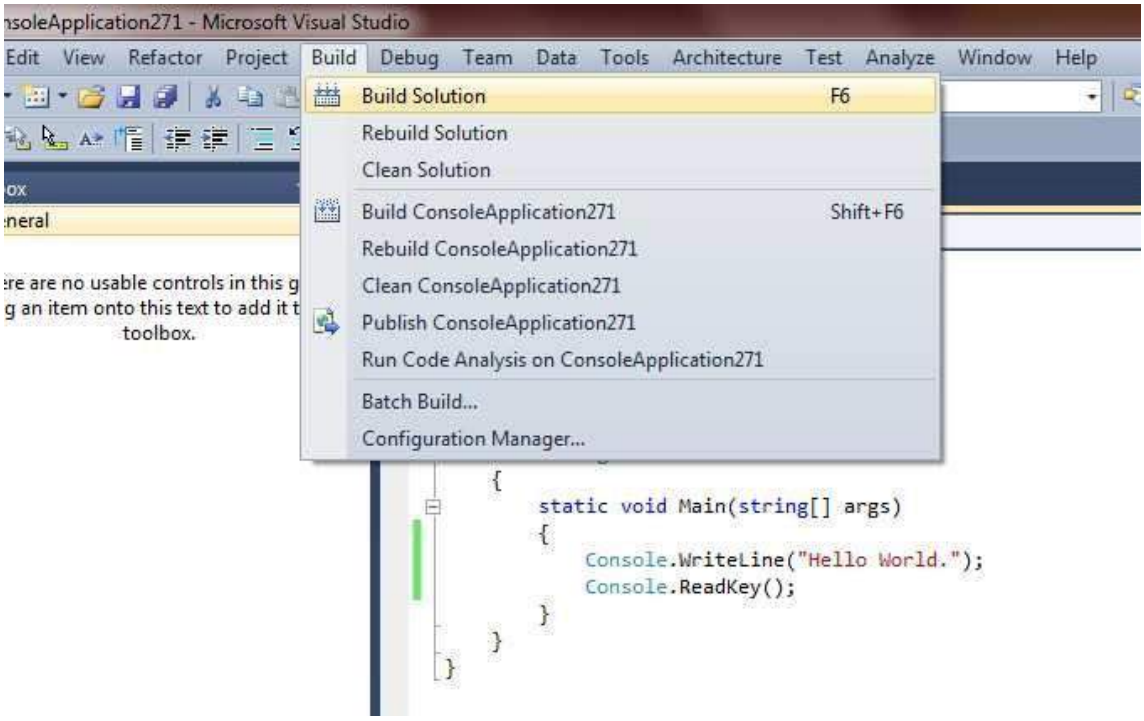
Objašnjenje: funkcija za unos nekog znaka sa tastature.

Sa ovakvim izmenama, naš kod izgleda ovako:

```
  using
System;
using
System.Collections.Generic;
using System.Linq; using
System.Text;
 namespace
ConsoleApplication271
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World.");
            Console.ReadKey();
        }
    }
}
```

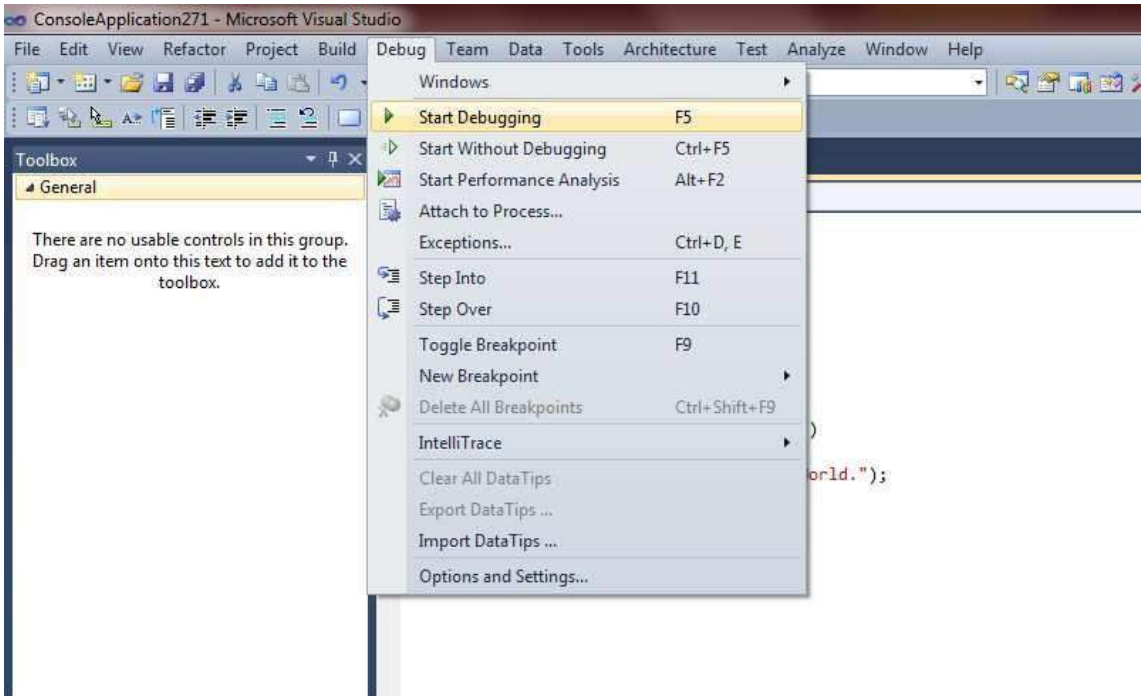
Da bismo realizovali u potpunosti naš program, potrebno je da ga izgradimo i kompajliramo.

Izgradnju radimo na sledeći način: Build -> Build Solution (kao na slici ispod):

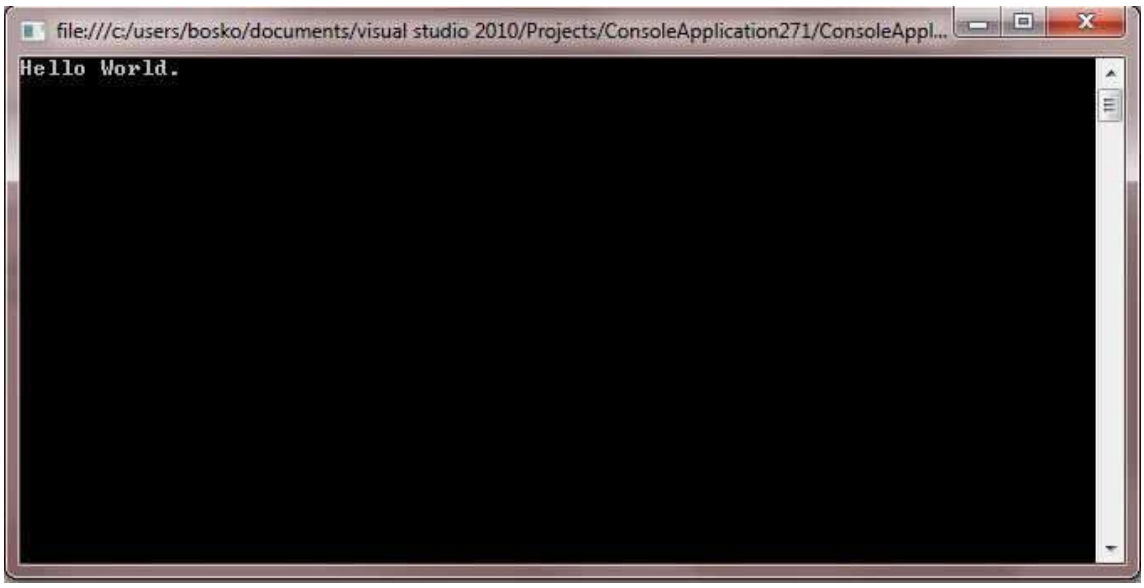


Preporučljivo je da se pre startovanja aplikacije obavezno pokrene debugging. To se realizuje na sledeći način:

Debug-> Start Debugging ili jednostavnim pritiskom na F5 taster.



Nakon ovoga ako izvršimo naš kod, u komand propmptu će se pojaviti tekstualna poruka: Hello World.



Ovim smo realizovali našu prvu konzolnu aplikaciju.

Nastavni čas 3 – Osnove jezika C#

Tipovi

Osnovna podela tipova je na:
–vrednosne (*value*) tipove
–referentne (*reference*) tipove
Vrednosni tipovi su:
–jednostavni tipovi (kao što su npr. `byte`, `int`, `long`, `float`, `double`)
–nabrojivi tipovi
–strukture
Referentni tipovi su: –klase
–interfejsi
–nizovi
–delegati
Svi tipovi (uključujući jednostavne kao što je `int`) su podtipovi `System.Object`

Razlike između vrednosnih i referentnih tipova:

Nekoliko bitnih razlika u osnovnim karakteristikama je između vrednosnih I referentnih tipova i to:

Razlike u alokaciji memorije:
vrednosni tipovi se čuvaju na steku, odnosno u okviru objekta, a ako su članovi referentnih tipovana čuvaju se na hipu
referentni tipovi se čuvaju na hipu Razlike
u sadržaju:
vrednosni tipovi sadrže podatak
referentni tipovi sadrže pokazivač (referencu) na podatak Razlike
u uništenju:
vrednosni tipovi se uništavaju odmah pošto se napusti oblast definisanosti
referentni tipovi se uništavaju pomoću sakupljača đubreta

Ugrađeni tipovi

Tip	Veličina (b)	.NET tip	Opis
byte	1	Byte	Vrednosti 0-255.
char	2	Char	Unicode karakteri.
bool	1	Boolean	Tačno ili netačno.

sbyte	1	SByte	Vrednosti od -128 do 127.
short	2	Int16	Vrednosti od -32,768 do 32,767.
ushort	2	UInt16	Vrednosti od 0 do 65,535.
int	4	Int32	Vrednosti od -2,147,483,648 do
			2,147,483,647.
uint	4	UInt32	Vrednosti od 0 do 4,294,967,295.
float	4	Single	Vrednosti od približno +/-1.5 * 10^{-45} do približno +/-3.4 * 10^{38}.
double	8	Double	Vrednosti od približno +/-5.0 * 10^{-324} do približno +/-1.8 * 10^{308} .
decimal	12	Decimal	Format zadate preciznosti do 28 cifara sa određenom pozicijom decimalne tačke.
long	8	Long	Celi brojevi od 9,223,372,036,854,775,808 do 9,223,372,036,854,775,807.
ulong	8	UInt64	Celi brojevi od 0 do 0xffffffffffffffff.

Promenljive i konstante

Promenljive

Promenljive su osnovni objekti koji se koriste u programima. Promenljiva u svakom trenutku svog postojanja ima vrednost kojoj se može pristupiti i koja se može pročitati i koristiti, ali i koja se (ukoliko nije traženo drugacije) može menjati.

Imena promenljivih su odredena identifikatorima. Za identifikator možemo postaviti proizvoljno ime, ali ne sme počinjati sa brojem niti sme biti ključna reč.

Posebno trebamo voditi računa da je Case Sensitive, odnosno osetljivo na velika i mala slova.

Primer inicijalizacije i dodele vrednosti promenljivoj:

```
int mojaIntPromenljiva = 5;
```

Konstante

Konstanta je promenljiva čija se vrednost ne može menjati. Konstanta se dobija stavljanjem ključne reči const ispred promenljive prilikom njene deklaracije i inicijalizacije.

```
const int broj = 100;
```

Ograničenja:

Kao konstanta se mogu navesti samo lokalna promenljiva ili polje klase.

Konstante moraju biti inicijalizovane prilikom deklaracije, i kada im se jednom dodeli vrednost, ona se ne može promeniti.

Takođe, konstanta se ne može inicijalizovati vrednošću neke promenljive, odnosno vrednost kojom se inicijalizuje konstanta mora biti dostupna u vreme kompajliranja.

Konstante su uvek implicitno statičke, stim što nije dozvoljeno navođenje ključne reči static u deklaraciji konstante.

Prednosti upotrebe konstanti se ogledaju u tome što se kod se lakše razume i modifikuje (promena se vrši samo na jednom mestu prilikom prepravljanja koda).

Naredbe

Naredbe u C# predstavljaju celovitu programsku instrukciju i završavaju se (;).

```
int x; x =
14; int z
= x;
```

Operatori

U programskom jeziku C# postoji više operatora i to pokazuje upravo njegovu snagu.

Operator je funkcija koju primenujemo nad vrednostima ili promenljivima da bismo dobili željeni rezultat.

Operator pridruživanja (dodele)

Najjednostavniji i najviše korišćeni operator je takozvani operator pridruživanja. Ako imate bilo kakav primer izraza, onda je poenta da se rezultat svih operacija u izrazu dodeljuje ili čuva u odgovarajućoj promenljivoj.

Primer: y = (m * x) + c;

Dakle, znak jednakosti predstavlja, u stvari, operator pridruživanja.

Aritmetički operatori

Aritmetičke operatore koristimo onda, kada želimo da izvršimo određenu matematičku operaciju nad jednim, dva ili više promenljivih, ili matematički rečeno operanada.

Inkrementiranje i dekrementiranje

Inkrementiranje predstavlja aritmetičku operaciju uvećanja promenljive za jedan, a dekrementiranje aritmetičku operaciju umanjenja promenljive za jedan.

NAPOMENA

Ispis x++ se naziva postincrement. To znači da kompajler uzima trenutnu vrednost promenljive x prilikom raznih izračunavanja i tek po završenim izračunavanjima inkrementira promenljivu x (uvećava za jedan).

Ispis ++x se naziva preincrement. To znači da kompajler prvo uveća vrednost promenljive za jedan i onda tako uvećanu vrednost promenljive x koristi u daljim izračunavanjima.

Operatori poređenja

Operatore poređenja koristimo onda, kada želimo da tok programa usmerimo zavisno od nekog uslova. Proučite sami tipove operatora poređenja narednih deset sekundi.

Redosled operatora

Što se tiče redosleda izvršavanja operatora zapamtite da su aritmetički operatori stariji, što znači da se pre izvršavaju, u odnosu na operatore poređenja. U primeru na slajdu prvo će se sabrati promenljive *a* i *b*, pa se tek onda proverava da li je taj zbir različit od promenljive *c.*

Logički operatori

Logički operatori se koriste za izvršavanje operacija Bulove algebre, kao što su endovanje, orovanje, negacija i slično.

Operator NOT

Operator *NOT* primenjen na neku logičku vrednost vraća istu u suprotno stanje.

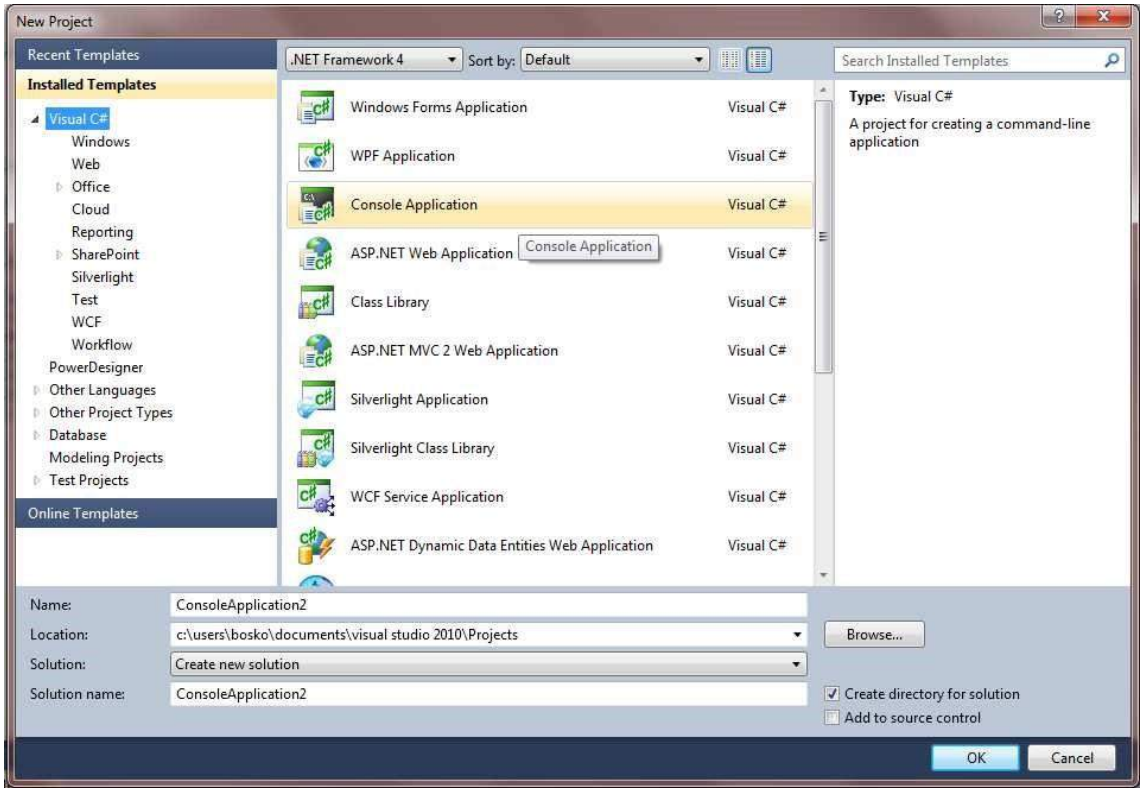
Tabela operatora u C#:

Category	Operators	Description	Associativity
Primary	( )	Precedence override	Left
	++	Post-increment	
	--	Post-decrement	
Unary	!	Logical NOT	Left
	+	Addition	
	-	Subtraction	
	++	Pre-increment	
	--	Pre-decrement	
Multiplicative	*	Multiply	Left
	/	Divide	
	%	Division remainder (modulus)	
Additive	+	Addition	Left
	-	Subtraction	
Relational	<	Less than	Left
	<=	Less than or equal to	
	>	Greater than	
	>=	Greater than or equal to	
Equality	==	Equal to	Left
	!=	Not equal to	
Conditional AND	&&	Logical AND	Left
Conditional OR	||	Logical OR	Left
Assignment	=		Right

Nastavni čas 4 – Konzolne aplikacije

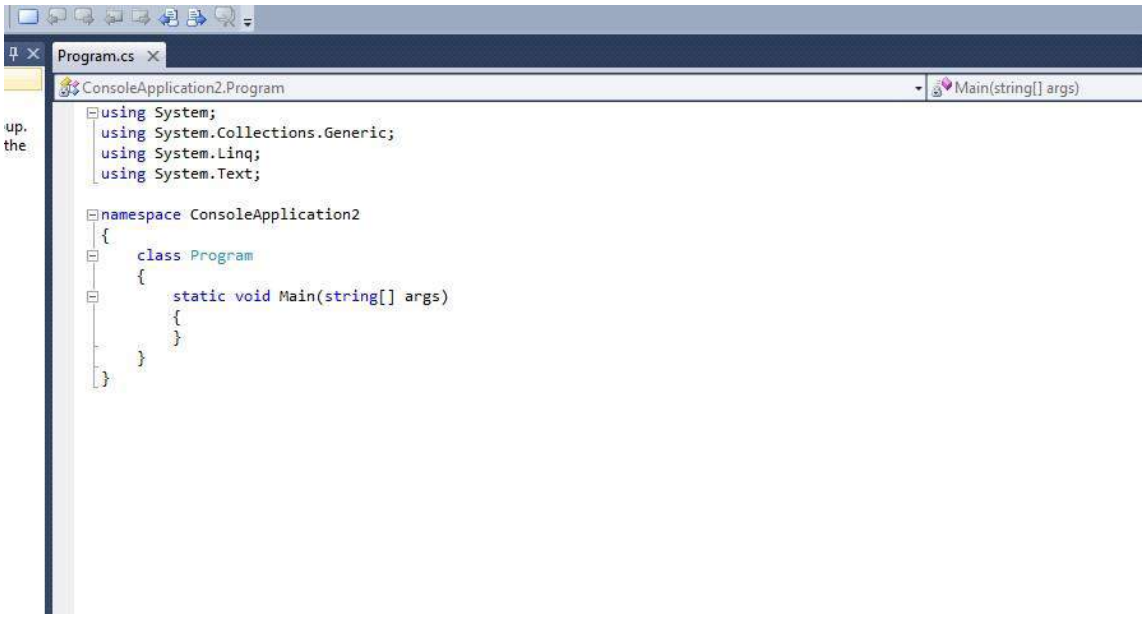
Postupak kreiranja konzolnih aplikacija

Da bismo kreirali novu konzolnu aplikaciju, potrebno je da nakon kreiranja novog projekta u opcijama odaberemo konzolnu aplikaciju kao na slici ispod.



Sva podešavanja oko imena i lokacije važe sa prethodnih vežbi.

Nakon potvrde na dugme ok, biće Vam kreiran osnovni kod konzolne aplikacije.



Ovaj kod je polazni osnov za kreiranje svih ostalih akcija u kodu.

Konzolna aplikacija – sabiranje stringova

Malo uanpređenija verzija programa Hello World je rad sa sabiranjem promenljivih stringovnog tipa I ispis Boolean vrednosti za zbir tih promenljivih.

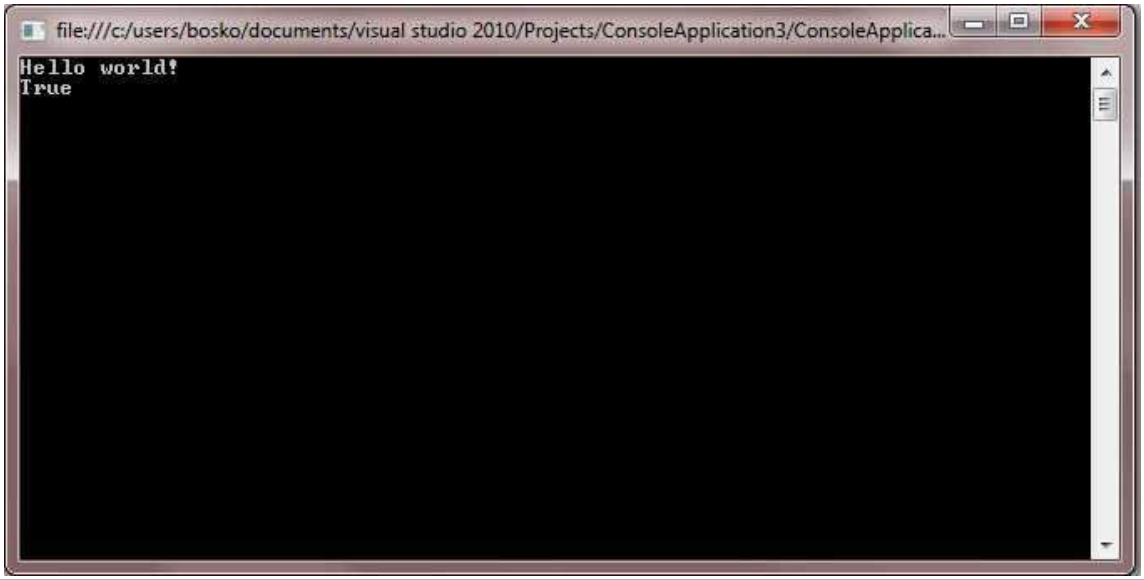
U ovom slučaju izvršićemo deklaraciju tri stringovne promenljive I dodeliti im vrednosti, a zatim ih sabrati. Na kraju ćemo izvršiti ispis rezultata pitanja da li je zbir tih promenljivih isti kao I tekst kojim ga upoređujemo.

```
using System;
using
System.Collections.Generic;
using System.Linq; using
System.Text;
 namespace
ConsoleApplication3
{
    class Program
    {
        static void Main(string[] args)
        {
            string a = "Hello "; // Deklarisanje i iniciranje promenljive a
string b = "world"; // Deklarisanje i iniciranje promenljive b
string c = "!"; // Deklarisanje i iniciranje promenljive c

            Console.WriteLine(a + b + c); // Konzolni ispis zbira stringova
Console.WriteLine(a + b + c == "Hello world!"); // Konzolni ispis boolean vrednosti

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

Rezultat:



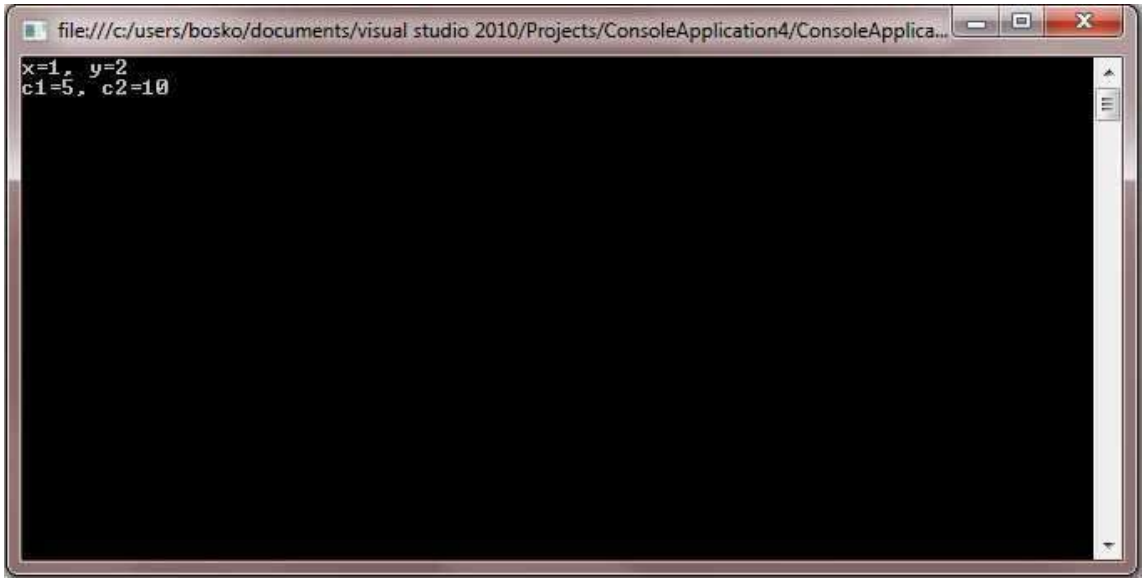
Konzolna aplikacija – operacije sa celim brojevima

Analogno radu sa stringovima, možemo se pozabaviti i drugim tipovima. Konkretno, u ovom primeru je operacija sa celim brojevima.

```
using System;
using
System.Collections.Generic;
using System.Linq; using
System.Text;
 namespace
ConsoleApplication4
{
    class Program
    {
        static void Main(string[] args)
        {             int x = 1;
int y = 2;             const int
c1 = 5;             const int c2
= c1 + 5;

            Console.WriteLine("x={0}, y={1}", x, y);
            Console.WriteLine("c1={0}, c2={1}", c1, c2);

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



Konzolna aplikacija – unos celog broja

U slučaju da želimo da unesemo neki ceo broj sa tastature, to možemo da uradimo na sledeći način:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
 namespace
ConsoleApplication5
{
    class Program
    {
        static void Main(string[] args)
        {
int a;
            Console.Write("Unesite ceo broj ");
            string str = Console.ReadLine();//Svaki unos iz konzole je u stringovnom
formatu!             a = int.Parse(str); // Parsiranje - string u int

            Console.WriteLine();
            Console.WriteLine("Broj koji ste uneli je {0}", a);
            Console.ReadLine();
        }
    } }
```

Napomena: Da bi se u potpunosti obezbedila funkcionalnost ovog programa, potrebno je obezbediti elemente koji će proveravati unos, odnosno neće dozvoliti nepravilne vrednosti. To se može uraditi dodatnim kodom koji će vršiti validaciju, ili try-catch mehanizmom. O svemu ovome će biti reči na narednim vežbama.

Nastavni čas 5 - Klase i objekti u Visual C#

Pre nego što se upustimo u složenu tematiku objašnjavanja klasa i objekata, bitno je da uradimo jedan mali pregled osnovnih razloga uvođenja objekto orijentisanog pristupa, onosno razlog postojanja klasa i objekata.

Konkretno, u proceduralnom programiranju, na osnovu koncepta izdovijio se niz nedostataka koji su zahtevali promene. Na primer, u proceduralnom programiranju, svaki podprogram može da pristupi svakom podatku, zatim, svaka promena u podacima može dovesti do neuspešnog izvršavanja podprograma. Još jedna bitna stvar koja je indirektno dovela do razmatranja je činjenica da je sa povećanjem veličine programa sve je teže vršiti izmene, a kod nije bilo moguće ponovno koristiti.

Uvođenjem objektno orijentisanog pristupa, rešavaju se ovi problemi, a pore toga ostvaruju se i sledeće ključne benefiti:

U objektno orijentisanom programiranju se polazi od objekata kojima želimo da manipulišemo, a ne od logike koja je potrebna za tu manipulaciju.
U realnom sistemu se identifikuju objekti i veze koje postoje između njih.

Definišu se tri osnovna koncepta objektno orijentisanog programiranja:

učaurenje
nasleđivanje
polimorfizam

Uvođenjem objektno orijentisanog pristupa, uvedeni su pojmovi klase i objekta.

Klase i objekti

Ako u najkraćem sažmemo predavanje br. 8 u delu objašnjenja pojma klasa i objekata, i pokušamo da odgovorimo na pitanje šta su to klase i objekti, to možemo uraditi na sledeži način:

Klasa je struktura podataka koju bismo trebali posmatrati kao novi tip, i ona prestavlja generičku definiciju objekata koji imaju zajeničku strukturu i ponašanje.

Nasuprot klasi, objekat je instanca klase i on se definiše kao entitet koji je sposoban da čuva svoja stanja i koji okolini stavlja na raspolagaje skup operacija preko kojih se tim stanjima pristupa.

Prema tome, objekat karakterišu:

- identitet - omogućava razlikovanje objekata među sobom
- ponašanje - dinamički aspekt objekta, definiše se metodama koje sadrži objekat
- stanje - statički aspekt objekta, definiše se podacima objekta i njegovim vezama sa drugim objektima u sistemu.

Kreiranje klase

Kada kreiramo neku sopstvenu klasu, potrebno je da definišemo: njena svojstva (properties) ili varijable kojima će dodeljivati vrednosti[1] i metode koje određuju njeno ponašanje.

Naredba za kreiranje nove klase počinje ključnom reči class, zatim dolazi naziv klase, zatim deklaracija svojstava (properties) koje se definišu za tu klasu, a zatim metode koje pripadaju klasi.

```
    public class mojaKlasa
    {
        //deklaracija i inicijalizacija svojstava ili varijabli
int mojaVarijabla;

        //metode
        public void mojaMetoda()
        {
...
        }
    }
```

Gornjim naredbama kreirali smo klasu mojaKlasa, i njoj pripadajuću celobrojnu varijablu mojaVarijabla kojoj će se unutar klase mojaKlasa dodjeljivati vrednosti.
U nastavku, još jedan primer kreiranja klase, ovoga puta, klasa Račun.

```
    public class Racun
    {
```

[1] Ako se radi o javnim varijablama, tada se koristi izraz “svojstvo” (eng. property), a ako se radi o privatnim varijablama koje će se koristiti samo u toj klasi, tada se koristi izraz “varijabla”.

```
        private long brojRacuna;
private decimal stanje;
private decimal provizija;
         public void Uplata(decimal
iznos)
        {
...
        }          public void
Isplata(decimal iznos)
        {              ...
}         public void
Prikazi()
        {
...
        }
    }
```

Kreiranje objekata

U prethonom delu smo definisali kreiranje klasa i vrlo je bitno da shvatimo da je klasa referentni tip. Samim tim, jasno nam je da se deklaracijom klase ne kreira instanca klase (objekat), već referenca.

Kreiranje objekata se realizuje na drugačiji način. Instanciranje klase, odnosno kreiranje objekta vrši se iz dva koraka:

- alokacija memorije se vrši pomoću operatora new.
- inicijalizacija objekta se vrši pomoću konstruktora kojim se alocirana memorija "pretvara" u objekat i objekat inicijalizuje.

Pristup članovima objekta ostvaruje se navođenjem punog kvalifikovanog imena člana.

```
    class Student
    {
        // deklaracija svojstava koja pripadaju
klasi         int Ocena;
        public void Main()
        {
            Student Nikola = new Student();[2]
            Nikola.Ocena = 3;[3]
        }
    }
```

[2] Instanciranje - na ovom mestu se kreira instanca Nikola koja pripada klasi Student.
[3] Dodela vrednosti - varijabli Ocena koja pripada instanci Nikola dodeljuje se vrednost 3.

Korišćenje ključne reči "this"

Ključna reč "this" odnosi se na trenutnu instancu neke klase (ili drugog objekta). "this" je skriveni pokazivač (pointer) na svaku nestatičku metodu neke klase.

Postoje tri slučaja upotrebe ključne reči this:

1. slučaj upotrebe: U svrhu kvalifikacije instance ako se ona zove isto kao i parametar koji se prosljeđuje nekoj metodi, pa tada "this" omogućava da se odredi na koju se vrednost se misli.

```
public void NasMetod (int varijabla)
{
    this.varijabla = varijabla;
}
```

2. slučaj upotrebe: U svrhu prosleđivanja trenutno aktivnog objekta kao parametra drugoj metodi

```
public void FirstMethod(OtherClass otherObject)
{
    otherObject.SecondMethod(this);
}
```

Ovaj primer uspostavlja dve klase; prva - klasa koja ima metodu FirstMethod(), I druga - klasa OtherClass koja ima metodu SecondMethod().

Ako unutar prve metode želimo da pozovemo metodu koja pripada drugoj klasi, tada tu drugu metodu moramo pozvati iz te druge klase. Tu klasu ne možemo koristiti preko njenog naziva, nego moramo proslediti parametar – varijablu koja je tipa OtherClass.

3. Slučaj upotrebe: korišćenjem indeksera

Primer korišćenja ključne reči this:
Ako this koristimo u svrhu određivanja čije svojstvo (property) će se koristiti:

```
label4.Text = this.ImePrezimeStudenta;
```

Gornjom naredbom u tekstu labele `label4` će se ispisati vrednost svojstva `ImePrezimeStudenta` koje pripada aktuelnoj klasi (u kojoj se nalazi ta metoda i naredba).

```
textBox1.Text = this.Ocena;
```

Gornjom naredbom u tekstu `textBox1` će se ispisati vridnost svojstva Ocena koje pripada aktuelnoj klasi (u kojoj se nalazi ta metoda i naredba).

Nastavni čas 6 – Klase i objekti u Visual C#

Metode

Metodama u programskom jeziku C# postižemo funkcionalnost nekog dela programa, odnosno njom izvršavamo neku akciju.
Osnovna sintaksa bilo kakve metode je:

<modifikator_pristupa>_<povratna_vrednost>_<ime_metode>(<argumenti_metoda>)
{
<telo_metode>
}

Argumenti metoda i povratne vrednosti

U programskom jeziku postoje različite varijacije metoda koje se mogu pojaviti u upotrebi, pa tako imamo nekoliko različitih varijanti:

Najjdenostavnija metoda je metoda koja ne vraća nikakvu vrednost, a ne prosleđuju joj se nikakvi parametri.

Primer je:

```
private void metodaBezArgumenataNeVracaVrednost()
        {
            UradiNesto();
        }
```

Važno je zapaziti da za ovakve metode mora biti definisana povratna vrednost na void.

Nešto složenija je metoda koja vraća nekakvu povratnu vrednost:

Primer je:

```
private String metodaBezArgumenataVracaStringovnuVrednost()
        {
            String nasaVrednost = "testVrednost";
return nasaVrednost;
        }
```

Sa druge strane, moguće je imati metodu koja ne vraća nikakvu vrednost ali kao argumente prima određene vrednosti:

```
        private void metodaSaArgumentimaNeVracaVrednost (String nasaVrednost, String
drugaVrednost)
        {
            nasaVrednost += drugaVrednost;

        }
```

Postoji mogućnost da nam je potrebno da metodi prosledimo argumente I da nam ta metoda vrati odgovarajuću vrednost.

U tom slučaju ćemo imati sledeće:

```
        private String metodaSaArgumentimaVracaVrednost (String nasaVrednost, String
drugaVrednost)
        {
            String sabraniStringovi;
            sabraniStringovi = nasaVrednost + drugaVrednost;
return sabraniStringovi;
        }
```

Primer upotrebe metoda možemo videti u prilikom realizacije osnovnih matematičkih operacija, gde pozivanjem metode matematičkeOperacije vršimo sabiranje brojeva 2 i 4 tako što svakoj metodi koja vrši određenu operaciju prosleđujemo kao argumente prvi i drugi broj, a kao rezultat rada metode prihvatamo vrednost.

```
public class MatematickeOperacije
    {          public String
matematickeOperacije()
        {             int
prviBroj = 2;             int
drugiBroj = 4;
            String zbir = "Zbir je " + sabiranje(prviBroj, drugiBroj) + ".";
            String razlika = "Razlika je " + oduzimanje(prviBroj, drugiBroj) + ".";
            String proizvod = "Proizvod je " + mnozenje(prviBroj, drugiBroj) + ".";
String kolicnik = "Kolicnik je " + deljenje(prviBroj, drugiBroj) + ".";
return zbir + "\n" + razlika + "\n" + proizvod + +"\n" + kolicnik;
        }          public int sabiranje(int prviBroj,
int drugiBroj)
        {             int zbir = prviBroj +
drugiBroj;             return zbir;
        }          public int oduzimanje(int prviBroj,
int drugiBroj)
        {             int razlika = prviBroj -
drugiBroj;             return razlika;
        }          public int mnozenje(int prviBroj,
int drugiBroj)
        {
            int proizvod = prviBroj * drugiBroj;
return proizvod;
```

```
        }          public double deljenje(int prviBroj,
int drugiBroj)         {
            double kolicnik = prviBroj / drugiBroj;
return kolicnik;
        }
    }
```

Konstruktori i destruktori

U prethodnim vežbama obrađena je sintaksa konstruktora u okviru metoda, ali nismo zašli "ispod haube" i nismo videli karakteristike konstruktora, njihove osnovne primene i nismo dali nikakve preporuke. Kako je sintaksa već obrađena, na nju se nećemo vraćati, ali ćemo konstruktore obraditi za potrebe razumevanja njihovog korišćenja.

Pre nego što budemo dali ilustrativni primer, bitno je da se zapamte sledeće činjenice:

Šta su konstruktori? Konstruktori su specijalne metode koje služe za inicijalizaciju objekata nakon njihovog kreiranja. Oni obezbeđuju da objekat ima dobro definisano početno stanje pre nego što se upotebi. Ako ne uspe inicijalizacija neće postojati objekat.

Za imenovanje konstruktora koristi se ime klase (metoda ima isto ime kao i klasa) iza koga slede zagrade.

Konstruktori nemaju povratnu vrednost, pa čak ni tipa void.

Postoje dve vrste konstruktora i to:

konstruktori instance (vrše inicijalizaciju objekata) statički

konstruktori (vrše inicijalizaciju klasa)

Kada se kreira objekat .NET kompajler, ukoliko nije eksplicitno naveden konstruktor, automatski generiše podrazumevani konstruktor.

Primer, kroz koji možemo da vidimo navedene činjenice je sledeći:

```
    class Datum
{
        private int godina;
private int mesec;
private int dan;
        // public Datum () { ... } default konstruktor koji se automatski generiše
pošto nije naveden konstruktor
    }
    class Test     {
static void Main()
        {
```

```
            Datum danas = new Datum(); //kreira se objekat i poziva konstruktor
...
        }
    }
```

Vrlo je bitno da zapamtite osobine podrazumevanih konstruktora zbog njegove dalje upotrebe u raznim Vašim programima (koristićete ih u svakoj klasi koju napravite).

Osobine podrazumevanog konstruktora su:

Podrazumevani konstruktor ne prima parametre.

Podrazumevani konstruktor implicitno inicijalizuje sva nestatička polja na njihove podrazumevane vrednosti i to: numerička polja (int, double,decimal) na nulu logička polja na false polja referentnog tipa na null polja tipa zapis tako da su svi elementi zapisa inicijlizovani na njihove podrazumevane vrednosti.

Modifikator pristupa je public.

Konstruktor može primiti jedan ili više parametara koji se koriste za inicijalizaciju polja.

Ako se u klasi deklariše bar jedan konstruktor, kompajler neće generisati podrazumevani konstruktor.

```
    public class Datum
    {
        public int godina, mesec, dan;
public Datum(int g, int m, int d)
        {
godina = g;
mesec = m;
dan = d;
        }
}     class
Test     {
        static void Main()
        {
            Datum danas = new Datum(2011,09,21);
        }
    }
```

Sva polja koja nisu inicijalizovana u korisnički definisanom konstruktoru zadržavaju svoju podrazumevanu inicijalizaciju.

```
    public class Datum
    {
        public int godina, mesec, dan;
public Datum(int g, int m, int d)
```

```
        {
godina = g;
mesec = m;
dan = d;
        }
}
    public class Primer
    {
        static void Main()
        {
            Datum danas = new Datum(2011, 09, 21);
            Console.WriteLine(danas.godina); // ispisuje 2010
Console.WriteLine(danas.mesec); //ispisuje 9
Console.WriteLine(danas.dan); // ispisuje 21
        }
    }
```

Za jednu klasu može se definisati više konstruktora.

NAPOMENA: lista parametara svakog od njih mora biti jedinstvena, ili po broju ili po tipu parametara (odnosno moraju imati različite potpise)

Na prethodnoj vežbi, u delu metoda, objašnjeno je preklapanje, tako da se na te primere nećemo ponovo vraćati, ali ćemo samo napomenuti da se može desiti da preklopljeni konstruktori sadrže isti kôd.

Inicijalizatorska lista omogućava da jedan konstruktor poziva drugi koji je deklarisan unutar iste klase čime se omogućava da se konstruktor implementira pozivanjem preklopljenog konstruktora. Sintaksa za ove navode je sledeća: Prilikom pravljenja konstruktora navodi se : iza kojih sledi ključna reč this, a u zagradi su navedeni parametri.

NAPOMENA: Inicijalizatorske liste se mogu koristiti samo kod konstruktora!

```
    public class Datum
    {
        public int godina, mesec,
dan;         public Datum()
: this(2010, 3, 30)
        {         }         public
Datum(int g, int m, int d)
        {
godina = g;
mesec = m;
dan = d;
        }
    }
```

Destruktori su metode koje su u potpunosti suprotne konstruktorima. Služe da inicirane objekte unište, i to im je osnovna namena. U velikoj meri desktruktore

nećete koristiti jer programski jezik C# radi pod VM koja obezbeđuje garbage collection, te samim tim vrši destrukciju objekata, ali nekada ćete imati potrebu da eksplicitno uništite objekat. U tom slučaju ćete upotrebiti destructor. Ukoliko nemate adekvatni konstruktor u klasi, prilikom aktiviranja destruktora, destruktor će uništiti instancirani objekat podrazumevanog konstruktora (setite se priče o podrazumevanim konstruktorima). Osnovne osobine destruktora su sledeće:

- Destruktori se ne mogu definisati na strukturama, koriste sa isključivo sa klasama.
- Klasa može imati samo jedan destruktor.
- Destruktor ne može se nasljeđuje ili preopterećuje.
- Destruktor se ne može pozvati. On se automatski poziva. Destruktor ne koristi modifikatore ni parametre.

Primer korišćenja destruktora je sledeći:

```
class Auto
{
    ~Auto()  // destructor klase Auto
    {
        // naredbe čišćenja
    }
}
```

Ako idemo malo dalje u razmatranje, možemo postaviti i drugačiji primer:

```
class PrvaKlasa
{
    ~PrvaKlasa()
    {
        System.Diagnostics.Trace.WriteLine("Desktruktor prve klase je
pozvan.");
    }
}  class DrugaKlasa :
PrvaKlasa
{
    ~DrugaKlasa()
    {
        System.Diagnostics.Trace.WriteLine("Desktruktor druge klase je
pozvan.");
    }
}  class TrecaKlasa :
DrugaKlasa
{
    ~TrecaKlasa()
    {
```

```
        System.Diagnostics.Trace.WriteLine("Desktruktor trece klase je
pozvan.");
    }
}
class TestirajDestruktore
{     static void
Main()
    {
        TrecaKlasa trecaKlasa = new TrecaKlasa();
    }
 }
```

Startovanjem ovog malog primera dobijamo sledeći izlaz:

```
Desktruktor trece klase je pozvan.
Desktruktor druge klase je pozvan.
Desktruktor prve klase je pozvan.
```

Iz njega jasno vidimo da destruktore ne pozivamo, ne prosleđujemo nikakve vrednosti, ali da oni vrše realizaciju bez obzira što mi to nismo uradili.

Šta u stvari destruktori rade prilikom realizacije?

Destruktor implicitno poziva metodu Finalize na osnovu bazne klase objekta. Ako pogledamo naš primer klase Auto, upotreba destruktora će se implicitno prevesti u sledeći kod:

```
    protected override void Finalize()
    {
try
        {
            // naredbe čišćenja
        }
finally
{
            base.Finalize();
        }
    }
```

Ovaj kod će biti jasniji posle završetka vežbe izuzeci.

Modifikatori pristupa

Vrlo je bitno da razlučimo šta su modifikatorui pristupa, koja su njihova ograničenja i shvatimo njihovu važnost.

Modifikatori pristupa u jeziku C# služe da odrede koje metode i promenljive članice drugih klasa određena klasa može da vidi i da koristi.
U daljem objašnjenju će ova rečenica dobiti plastični prikaz.

Public

Modifikatorom public svi elementi klase koji su njim modifikovani su vidljivi svim metodama svih klasa.

Primer upotrebe je:

```
    public class ModifikatoriPristupa
    {
        public int celobrojnaVrednost = 1;
private void pristupiPoljuIzKlase()
        {
            celobrojnaVrednost++;
        }
    }
```

U ovom slučaju upotrebe iz bilo koje klase je moguće pristupiti ovom polju.

Private

Modifikatorom private je članovima klase moguće pristupiti samo metodama iz klase

Primer upotrebe je:

```
    public class ModifikatoriPristupa
    {
        private int celobrojnaVrednost = 1;
private void pristupiPoljuIzKlase()
        {
            celobrojnaVrednost++;
        }
    }
```

Nemoguće je pristupiti polju izvan klase, odnosno nije moguće uraditi sledeće:

```
    public class ModifikatoriPristupa
    {
        public int celobrojnaVrednost = 1;
private void pristupiPoljuIzKlase()
        {
            celobrojnaVrednost++;
        }
    }      public class
TestKlasa
    {         private void
pristupiPoljuIzKlase()
        {
```

```
            ModifikatoriPristupa instance = new ModifikatoriPristupa();
instance.celobrojnaVrednost = 1;
        }
    }
```

Protected

Modifikatorom protected svi članovi klase dostupni su samo metodama klase I metodama klasa koje su izvedene iz klase.

Internal Članovi klase su dostupni svim metodama svih klasa iz programskog sklopa u kome je klasa.

Protected Internal

Članovi klase su dostupni svim metodama klasa, metodama izvedenim iz klase I klasama programskog sklopa u kome je klasa.

Enkapsulacija

Enkapsulacija objekata je jedan od osnovnih koncepata objektno orijentisanog programiranja I predstavlja dodatnu apstrakciju kojom se “sakrivaju” detalji implementacije objekta.

Postoje dva bitna aspekta enkapsulacije:

objedinjavanje podataka i funkcija u jedinstven entitet (klasa) kontrola

mogućnosti pristupa članovima entiteta (modifikatori pristupa)

NAPOMENA: Posle ovog dela Vam mora ostati ova činjenica urezana duboko u pamćenje: direktan pristup podacima je potpuno nepotreban i nepoželjan!

Objedinjavanje podataka i funkcija u jedinstven entitet se ostvaruje se pomoću klasa, a određuje se granicom entiteta.

Kontrola mogućnosti pristupa članovima entiteta ostvaruje se navođenjem modifikatora pristupa. Uvođenjem modifikatora pristupa omogućava se razdvajanje klase na javni deo koji čine članovi koji su označeni sa modifikatorom pristupa public (pristup nije ograničen) i privatni deo koji čine članovi koji su označeni sa modifikatorom pristupa private (mogu mu pristupiti samo članovi klase).

```
        public void Uplata(decimal iznos)
        {
```

```
            stanje = stanje + (iznos*(1-provizija));
        }          public void Isplata(decimal
iznos){...}         public void Prikazi(){...}
         public long brojRacuna;
public decimal stanje;
public decimal provizija;
        }

...
    Racun racun1 = new Racun();
racun1.Uplata(10000);
racun1.stanje = 150000;
```

Na osnovu principa objektno orijentisanog programiranja I dobre programerske prakse, preporuka je da podaci objekta treba da se nalaze u privatnom delu. Naravno, ukoliko to ne želite da uradite ne morate, ali tako sami sebi usložnjavate održavanje, a samim tim I mogućnost nastanka bug-ova.

Od mesta deklaracije određenog člana zavisi i vrsta modifikatora pristupa koji se može koristiti. Ukoliko nije naveden, određuje se podrazumevani modifikator:

- za imenovani prostor nije dozvoljeno navođenje modifikatora, jer se podrazumeva da su javni
- tipovi koji se deklarišu unutar imenovanog prostora mogu biti public ili internal. Podrazumevani modifikator je internal.
- članovi klase mogu da imaju bilo koji od navedenih pet modifikatora. Podrazumevani modifikator je private.

članovi zapisa mogu biti public, internal ili private. Podrazumevani modifikator je private. članovi interfejsa ne mogu imati modifikatore pristupa. Implicitno su public. članovi nabrajanja ne mogu imati modifikatore pristupa. Implicitno su public.

Domen iz koga se može pristupiti određenom tipu nikad ne sme biti uži od domena iz koga se može pristupiti članu koji je deklarisan unutar tog tipa.

```
namespace primer
{
    internal class Racun
    {
        public decimal stanje; //greška
    } }
```

Postoje dva razloga:

- omogućavanje kontrole korišćenja - Objekat se može koristiti isključivo preko javnih metoda.
- smanjenje uticaja promena - Ukoliko su detalji implementacije objekta privatni mogu se promeniti, a da te promene ne utiču direktno na korisničke objekte (koje jedino mogu da pristupe javnim metodama).

```
       public class Racun {
public decimal DajStanje()
    {
        return stanje;
    }
    public void Prikazi(){...}
private decimal stanje;
}

public class Klijent {

    public void Prikazi()
    {
        Console.WriteLine( “Klijent ...” );
        Console.WriteLine( string.Format(“Stanje: {0}” racun.DajStanje() );
}

    private Racun racun;
}
```

Ovim promena ne utiče na korisnika klase.

```
public class Racun {
public decimal DajStanje()
```

Nastavni čas 7 – Logičke strukture

U programskom jeziku postoji više naredbi koje mogu da uslove različite tokove i izvrše grananja. Takođe, postoje i različite petlje koje će omogućiti višestruko prolaženje kroz kod kada je to potrebno. U narednom odeljku dat je prikaz tih struktura.

Naredbe grananja

U programskom jeziku C# postoji nekoliko naredbi koje pod određenim uslovima mogu da realizuju grananje i to:

If

Jednostavna if naradba vrši proveru nekog uslova i omogućava realizaciju bloka naredbi ako je uslov tačan (true).

Primer:

Ako je broj veći od 0 inkrementiraj brojač.

```
if(x > 0)
{

brojac++;             }
```

if – else

If – else naredba se koristi ako imamo slučaj u kome je potrebno uraditi jednu stvar za vrednost uslova, u suprotnom je potrebno uraditi drugu stvar.

Primer:

Ako je broj veći od 0 inkrementiraj brojač, u suprotnom dekrementiraj brojač.

```
            if (x > 0)
brojac++;
else
brojac--;
```

Kombinacijom višestrukih uslova može se dobiti nekoliko varijacija I to:

if – else – if

Primer:

Ispiši poruku da li je broj manji od 6, u opsegu od 6 do 13 ili je veći od 13.

```
            if (x < 6)
            {
                System.Console.WriteLine("Manji od sest.");
} else if (x >= 6 && x <= 13) {
                System.Console.WriteLine("U opsegu je od 6 do 13");
} else
            {
                System.Console.WriteLine("Veci je od 13.");
            }
```

ugnježdeni if – else

Primer:

Ako je broj veći od 0, a ako je brojač manji od 2, brojač postavi na 6, a ako je broj manji od 2, brojač postavi na nulu.

```
            if (x > 0)
{
                if (brojac < 2)
                {
                    brojac = 6;
                }
else
{
                    brojac = 0;
                }
            }
```

switch

Ukoliko imamo jedan uslov po kome imamo samo jednu opciju koja je ispunjena, u tom slučaju je idealan izbor korišćenja naredbe switch.

Primer:

Na osnovu prosledjenog broja od 1 do sedam ispisati naziv dana u nedelji.

```
public void primerSwitch(int dan)
        {

            System.Console.WriteLine("Danas je " + dan + " dan u nedelji");
             switch
(dan)             {
case 1:

Console.WriteLine("Ponedeljak");
break;                  case 2:

Console.WriteLine("Utorak");
break;                  case 3:

Console.WriteLine("Sreda");
break;                  case 4:

Console.WriteLine("Cetvrtak");
break;                  case 5:

Console.WriteLine("Petak");
break;                  case 6:

Console.WriteLine("Subota");
break;                  case 7:
```

```
Console.WriteLine("Nedelja");
break;                  default:
                    Console.WriteLine("Ima samo 7 dana");
break;
            }

        }
```

Pored naredbi grananja, postoje i mehanizmi grananja (try - catch, overriding, overloading), ali će o njima biti reči u narednim predavanjima.

Petlje

Petlje su kontrolne strukture pomoću kojih možemo da izvršimo višestruko ponavljanje jednog dela koda. Petlje se koriste u nekoliko karakterističnih situacija: prilikom kontrola toka, rada sa nizovima, rada sa kolekcijama, iteracijama, čekanje na ispunjenje uslova, itd.

U programskom jeziku C# postoji nekoliko petlji koje se mogu realizovati i to:

for

For petlja se sastoji od naredbe for, tri dela for naredbe u običnoj zagradi, a u vitičastoj zagradi se nalaze naredbe koje se vrte u okviru petlje. Tri dela koja for petlja ima su odvojena pomoću oznake ;. Prvi deo for petlje od otvorene obične zagrade do prve; služi za inicijalizaciju brojaèa. Drugi deo, koji se nalazi između dve oznake ; , je uslov. Treći deo, od ; do zatvorene obične zagrade, je inkrement odnosno decrement brojača. U prvom delu petlje, koji služi za inicijalizaciju brojača, se nalazi deklaracija brojača (u ovom slučaju int i) i njegova inicijalizacija na početnu vrednost (u ovom slučaju na 0). U drugom delu je uslov. Dokle god je uslov ispunjen, odnosno dokle god je izraz između dve oznake ; true, petlja će da se izvršava (vrti).

```
for (int i = 0; i < x; i++)
            {
                x = i * i;
            }
```

Specijalni slučaj for petlje je kada se u bloku naredbi nalazi samo jedna naredba koja se ponavlja. U tom slučaju mogu da se izostave oznake za blok (otvorena i zatvorena vitičasta zagrada), ali to nije dobra programerska praksa jer otežava održavanje koda.

```
for (int i = 0; i < x; i++)
x = i * i;
```

U slučaju da je brojač neka promenljiva koja je prethodno definisana, moguće je da se u okviru for petlje, izostavi deo za definiciju brojača.

```
 int i = 0 for (; i <
niz.length; i++)
            {
                x = i * i;
            }
```

Brojač može da se uvećava u delu bloka naredbi (ili da se umanjuje), i u toj situaciji nije neophodno imati inkrement (decrement) u delu for naredbe.

```
for (int i = 0; i < x; )
            {
                x = i * i++;
            }
```

Specijalna verzija for petlje je izostavljen uslov. Uslov postoji, ali nije u delu koji je predviđen u for petlji, već se nalazi u samom telu petlje.

```
for (int i = 0; ; i++)
{                   if (i <
x)
break;                   x
= i * i;             }
```

Specijalni slučaj for petlje je i kada se petlja vrti u beskonačno. Moguće je kreirati beskonačnu for petlju tako što se izostave sva tri dela for petlje.

```
for (; ; )
{
                //beskonacna petlja
            }
```

foreach

Naredba foreach je nova naredba u porodici c jezika I služi da obezbedi iterativni pristup svim elementima niza ili kolekcije.

Sintaksa je sledeća:

```
for (tip identifikator in izraz )
            {
naredba
            }
```

Primer:
Popunjavanje niza I štampanje elemenata niza.

```
int[] intniz = new int[5];
            int[] intniz2 = new int[3];
             for (int i=0; i<intniz2.Length; i++
)
            {
                intniz2[i] = i + 5;
            }
             foreach (int i in
intniz)
            {
                Console.WriteLine(i);
            }

            foreach (int i in intniz2)
            {
                Console.WriteLine(i);
            }
```

while

While petlja je petlja koja je specijalno optimizovana za prolazak kroz petlju dokle god je neki uslov ispunjen.

Primer:

```
int i = 0;
            while (i < niz.Length)
            {
                niz[i] = i *
i;                  i++;
}
```

do-while

Razlika između do – while petlje i svih ostalih petlji je u tome što se do – while petlja uvek izvršava bar jednom. Ispitivanje uslova ove petlje je na kraju.

Primer:

```
            int i = 0;
            int[] niz = new
int[5];             do
{
                niz[i] = i * i;
i++;
            }
```

```
        while (i < niz.Length);
```

Pored naredbi grananja i mehanizama grananja, postoje i naredbe koje utiču na tok programa.

Naredba break spada u naredbe grananja zbog toga što prekida tok izvršavanja programa u određenom trenutku i prosleđuje na neko drugo mesto nastavak izvršavanja programa

Naredba continue prekida započeti tok i vraća petlju na početak, pre njenog logičkog kraja. Naredba return prekida tok neke metode i vraća rezultat pre njenog kraja.

Nastavni čas 8 – Nasleđivanje, interfejsi i apstraktne klase

Nasleđivanje

Class Inheritance ili nasleđivanje je koncept objektno-orijentisanog programiranja koji omogućava da se na osnovu postojeće klase izvede nova klasa. Nova izvedena klasa nasleđuje sve članove bazne klase.

Bitne činjenice koje se moraju shvatiti su da se nasleđivanjem se ostvaruje veza (odnos) između objekata i to:

is veza bazna klasa izvedena klasa koja nasleđuje baznu klasu.

Napomena:

Treba razlikovati nasleđivanje klasa i nasleđivanje interfejsa (klasa koja nasleđuje interfejs mora da implementira sve navedene funkcije).

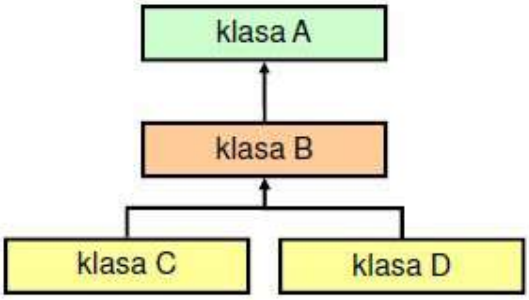
Nasleđivanje može biti direktno i indirektno.

Na slici ispod prikazan je dijagram na kom se lako može objasniti nasleđivanje:

Klasa C direktno nasleđuje klasu B, a indirektno klasu A

Sve promene nad baznom klasom (klasa A) se automatski odražavaju nad izvedenim klasama (klasa B, klasa C i klasa D). Obrnuto ne važi.

Jednu baznu klasu može da nasledi više izvedenih klasa Broj izvedenih klasa za jednu baznu klasu nije ograničen.
Iz klase B su izvedene klasa C i klasa D



U objektno orijentisanom programiranju postoji Jednostruko nasleđivanje i višestruko nasleđivanje. Za jenostruko nasleđivanje izvedena klasa direktno nasleđuje jednu baznu klasu, dok za višestruko nasleđivanje izvedena klasa direktno nasleđuje dve ili više baznih klasa.
U programskom jeziku C# nije dozvoljeno višestruko nasleđivanje.

Svaka izvedena klasa može dalje biti bazna klasa. Grupa klasa koje su povezane nasleđivanjem formiraju strukturu koja se naziva hijerarhija klasa pri čemu su klase na višim nivoima opštije, dok su one na nižim nivoima u hijerarhiji specifičnije.

Što se tiče dubine hijerarhije preporuka je da broj nivoa ne bude veći od sedam Ukoliko nije eksplicitno navedena bazna klasa podrazumeva se System.Object.

Sintaksa za nasleđivanje je sledeća:

```
class IzvedenaKlasa: BaznaKlasa
{...}
```

Izvedena klasa nasleđuje sve članove bazne klase osim konstruktora i destruktora.
Javni članovi bazne klase su implicitno javni članovi izvedene klase. Takođe, izvedena klasa može imati nove članove.
Primer:

```
    public class Racun
    {         public void Uplata(decimal
iznos){...}         public void
Isplata(decimal iznos){...}         private
long brojRacuna;         private decimal
stanje;
```

```
    }
    public class TekuciRacun: Racun
    {
        private decimal provizija; //novi
clan     }     ...
    TekuciRacun tekuci = new TekuciRacun();
    Tekuci.Uplata(100);
```

Nasleđivanje ne podrazumeva da će izvedena klasa imati pristup svim članovima bazne klase. Privatni članovi bazne klase, iako su nasleđeni, oni su dostupni isključivo članovima bazne klase (ukoliko Vam je ovo nejasno, vratite se na modifikatore pristupa u prethodnim vežbama).

Primer:

```
    public class Racun
    {         public void Uplata(decimal
iznos){...}         public void
Isplata(decimal iznos){...}         private
long brojRacuna;         private decimal
stanje;
    }
    public class TekuciRacun: Racun
    {
        public void Prikazi();
        {
            Console.WriteLine(“Stanje:”, stanje); //greška
        }
    }
```

Članovi bazne klase sa modifikatorom protected su jedino dostupni unutar bazne klase i direktno i indirektno izvedenim klasama.
Na osnovu principa objektno orijentisanog programiranja i dobre programerske prakse, treba težiti da sva polja klase treba da imaju modifikator pristupa private, a za svako od njih navesti svojstvo sa modifikatorom pristupa protected.

Primer:

```
    public class Racun
    {
...
        protected decimal stanje;
    }     public class
TekuciRacun: Racun
    {
        public void Prikazi()
        {
            Console.WriteLine(“Stanje:”, stanje); // OK
        }
```

```
    }
```

Metode izvedene klase ne mogu da pristupe članovima bazne klase preko reference. Primer:

```
    public class Racun
    {         ...
protected decimal stanje;
    }
    public class TekuciRacun: Racun
    {
        public void Prikazi(Racun racun)
        {
            Console.WriteLine("Stanje:", racun.stanje); // greška
        }
    }
```

Dostupnost izvedene klase je uslovljena dostupnošću bazne klase - ukoliko je bazna klasa privatna, izvedena klasa ne može biti javna.

Za poziv konstuktora bazne klase iz konstruktora izvedene klase se koristi ključna reč base. Prvo se izvršava konstruktor bazne klase. Navođenje inicijalizatora konstruktora nije obavezno (ukoliko se ne navede, poziva se konstruktor (bez parametara) bazne klase). Konstruktor bazne klase, implicitno poziva konstruktor klase System.Object

Primer:

```
    public class BaznaKlasa
    {
        public BaznaKlasa() {...}
    }
    public class IzvedenaKlasa: BaznaKlasa
    {
        public IzvedenaKlasa(): base() { ... }
}
```

Interfejsi

Interfejs predstavlja "ugovor" kojim se garantuje da će se klasa, koja je nasledila taj interfejs, ponašati na određeni način. Dakle, klasa garantuje da prodržava metode, svojstva (properties), događaje (events) i indeksere nekog interfejsa.

Sintaksno interfejs prestavlja klasu koja sadrži samo apstraktne metode. Sintaksa za implementaciju interfejsa izgleda ovako:

Primer:

```
class PrimerKlasa : PrimerInterface
{ ...
}
```

Kada klasa implementira interfejs ona mora da implementira sve njene metode!

Moguće je da jedna klasa implementira više interfejsa.
NAPOMENA: višestruko nasleđivanje klasa nije dozvoljeno u C#, ali jeste višestruko nasleđivanje interfejsa!

Primer:

```
using System;
 interface
IBazniInterface
{     void
BazniInterfaceMetod();
}  interface IMojInterface :
IBazniInterface
{
    void MetodaZaImplementaciju();
}  class InterfaceImplementer :
IMojInterface
{     static void
Main()
    {
        InterfaceImplementer iImp = new InterfaceImplementer();
iImp.MetodaZaImplementaciju();
iImp.BazniInterfaceMetod();
    }      public void
MetodaZaImplementaciju()
    {
        Console.WriteLine("MetodaZaImplementaciju() je pozvana.");
    }      public void
BazniInterfaceMetod()
    {
        Console.WriteLine("BazniInterfaceMetod() je pozvan.");
    }
}
```

Apstraktne klase

Kada uređujete klase u hijerarhiju nasleđivanja, obično su “više” klase apstraktnije i uopštenije, dok su “niže” klase konkretnije i specifičnije. Često polazna klasa i nema

instance, već služi kao izvor informacija koje podklase koriste. Ovakve klase se nazivaju apstraktnim i deklarišu se pomoću ključne reči abstract.

Definicija apstraktne klase izgleda ovako:

```
        public abstract class ApstrakntaKlasa
    {
            ...
      }
```

Apstraktne klase ne mogu da imaju objekte. Ukoliko pokušate da kreirate instancu apstraktne klase, izazvaćete grešku kompajlera. Međutim, one mogu da sadrže sve što i normalne klase, uključujući klasne i objektne promenljive i metode sa bilo kojim nivoom zaštite.

Primer:

```
using System;
 namespace
ApstraktneKlase
{      public interface
IRacun
    {         void IsplatiSaRacuna(double
iznos);         void UplatiNaRacun(double
iznos);         string
VratiPodatkeORacunu();         double
VratiStanje();
    }      public abstract class AbstractRacun :
ApstraktneKlase.IRacun
    {
        private double stanje;
private string brojRacuna;
        public AbstractRacun(string brojRacuna) : this(brojRacuna, 0) { }

        public AbstractRacun(string brojRacuna, double pocetnoStanje)
        {
            this.brojRacuna = brojRacuna;
this.stanje = pocetnoStanje;
        }
        public double VratiStanje()
        {
            return stanje;
        }          public void
UplatiNaRacun(double iznos)
```

```
        {
            stanje += iznos - ProvizijaNaUplatu(iznos);
        }          public void
IsplatiSaRacuna(double iznos)
        {             stanje -= iznos +
ProvizijaNaIsplatu(iznos);
        }
        protected abstract double ProvizijaNaUplatu(double iznos);
protected abstract double ProvizijaNaIsplatu(double iznos);
         public virtual string
VratiPodatkeORacunu()
        {             string podaci = "Racun broj: " + brojRacuna + "\nIznos na
racunu: " + stanje;             return podaci;
        }

        public class TekuciRacun : AbstractRacun
        {
            public TekuciRacun(string brojRacuna) : base(brojRacuna) { }
             public TekuciRacun(string brojRacuna, double pocetnoStanje)
: base(brojRacuna, pocetnoStanje) { }
             protected override double ProvizijaNaUplatu(double
iznos)
            {
                return 0;
            }

            protected override double ProvizijaNaIsplatu(double iznos)
            {
                double obracunataProvizija = 100 + (iznos * 3 / 100);
return obracunataProvizija;
            }

            public override string VratiPodatkeORacunu()
            {
                return base.VratiPodatkeORacunu() + "\nTip racuna: TEKUCI";
            }
        }          public class DevizniRacun :
AbstractRacun
        {             public DevizniRacun(string brojRacuna) :
base(brojRacuna) { }             public DevizniRacun(string brojRacuna,
double pocetnoStanje) : base(brojRacuna, pocetnoStanje) { }
            protected override double ProvizijaNaUplatu(double iznos)
            {
                double obracunataProvizija = 100 + (iznos * 5 / 100);
return obracunataProvizija;
            }

            protected override double ProvizijaNaIsplatu(double iznos)
            {
                double obracunataProvizija = 100 + (iznos * 5 / 100);
return obracunataProvizija;
            }
```

```
            public override string VratiPodatkeORacunu()
            {
                return base.VratiPodatkeORacunu() + "\nTipRacuna: DEVIZNI";
}
        }

          public static void Main(string[]
args)
        {
            AbstractRacun[] racuni = new AbstractRacun[2];
racuni[0] = new TekuciRacun("392-411232-417", 0);
racuni[1] = new DevizniRacun("396-411232-417", 0);
for (int i = 0; i < 2; i++)
            {
                Console.WriteLine("Uplata na " + i + ". racun u iznosu od 1000");
                racuni[i].UplatiNaRacun(1000);
                Console.WriteLine("Podaci o racunu posle uplate: \n"
+ racuni[i].VratiPodatkeORacunu());
Console.WriteLine();
            }

        }
    }
}
```

Nastavni čas 9 – Polimorfizam i overriding

Polimorfizam

Sposobnost promenljive da referencira objekte različitih tipova i da automatski poziva odgovarajuću metodu objekta koji se referencira se naziva polimorfizam.
Polimorfizam se zasniva na sledećem konceptu: metoda koja je deklarisana u baznoj klasi može da se implementira na više različitih načina u različitim izvedenim klasama.

Primer:

```
OsnovnaKlasa promenljiva = new IzvedenaKlasa();
```

Promenljiva tipa bazna klasa može da referencira instance direktno ili indirektno izvedenih klasa.
Polimorfizam se u jeziku C# najčešće postiže upotrebom virtuelnih metoda.

Za deklaraciju virtuelne metode se koristi ključna reč virtual.

Primer:

```
public class Racun
{     public virtual void Prikazi()
{...}
}
```

Prilikom definisanja, virtuelne metode se moraju implementirati, ukoliko to ne učinite, napravićete grešku.

Primer:

```
public class Racun
{     public virtual void Prikazi();
//greška }
```

Privatne i statičke metode se ne mogu definisati kao virtuelne. Ukoliko ne vodite računa o tome da virtuelne metode nisu modifikovane kao public, napravićete grešku.

Primer:

```
public class Racun
{     private virtual void Prikazi();
//greška }
```

Metode nisu implicitno virtuelne iz sledećih razloga:

performanse - potrebno je vreme da bi se odredilo koju od reimplementiranih metoda treba pozvati, što najčešće ne utiče u velikoj meri na performanse. Veći problem je nemogućnost optimizacije koda prilikom kompajliranja. Za nevirtuelne metode ova informacija je dostupna u vreme kompajliranja, što znači da se prilikom kompajliranja

mogu sprovesti određene optimizacije (npr. inline). dizajn

metode klase koje su namenjene za internu upotrebu i koje se odnose isključivo na dizajn date klase ne treba da se reimplementiraju u izvedenim klasama, pa samim tim ne treba ni da budu virtuelne.

Overriding

Overriing je uslovno jedan od osnovnih principa OO programiranja. • Proces implementacije virtuelne metode u izvedenoj klasi se naziva overriding.

Najbanalnije objašnjenje overridinga je nadjačavanje osnovne metode u baznoj klasi implementiranom metodom u izvedenoj klasi.

Za definisanje overriinga koristi se ključna reč override.

Primer:

```
public class Racun
```

```
{     public virtual void Prikazi()
{...}
} public class TekuciRacun:
Racun
{     public override void Prikazi()
{...}
}
```

Virtuelna i reimplementirana metoda moraju biti indetične, moraju imati:
isti naziv, isti
modifikator pristupa, isti
tip rezultata i iste
parametre.

Koja metoda će biti pozvana, metoda bazne ili izvedene klase određuje se na osnovu tipa instance koju promenljiva referencira, a ne na osnovu tipa same promenljive.

Primer:

```
public class Racun {
    public virtual void Prikazi() {
        Console.WriteLine("Racun: {0}", brojRacuna);
    } } public class TekuciRacun:
Racun {     public override void
Prikazi() {
        Console.WriteLine("Tekuci racun: {0}", brojRacuna);
    }
}
Racun racun = new
Racun(); racun.Prikazi();
racun = new
TekuciRacun();
racun.Prikazi();
```

Reimplementirana metoda se takođe može reimplementirati, reimplementirana metoda je implicitno virtuelna i ne može se eksplicitno tako deklarisati.

Nastavni čas 10 - Strukture, enumeratori, referenciranje i konverzija

Strukture podataka

Struktura (engl. struct) je jednostavan korisnički definisan tip, lakša alternativa klasi. Strukture su slične klasama po tome što mogu imati konstruktore, svojstva, metode, polja, operatore, ugnježdene tipove i indeksere.

Međutim, postoje i značajne razlike između klasa i struktura.

Na primjer, strukture ne podržavaju naasleđivanje, desturktore, a najvažnija razlika je u tome što su klase referentni tipovi, dok su strukture vrednosni tipovi. Samim tim, primena struktura je odlična za predstavljanje objekata kojima nije potrebna semantika referenci.

Definisanje struktura

Sintaksa za deklarisanje structure vrlo je slična sintaksi za kreiranje klase:

```
[atributi] [modifikator pristupa] struct identifikator [:lista interfejsa]
{ članovi }
```

Kreiranje struktura

U primeru ispod kreirana je struktura koja ilustruje njenu upotrebu.

```
using System;
using System.Collecti-ons.Generic;
using System.Text;
  namespace
CreatingAStruct
{     public struct
Location
    {
        private int xVal;
private int yVal;
         public Location(int xCoordinate, int
yCoordinate)
        {             xVal =
xCoordinate;             yVal
= yCoordinate;
        }
        public int x
        {             get {
return xVal; }             set {
xVal = value; }
        }
public int y
        {             get {
return yVal; }             set {
yVal = value; }
        }

        {                   public void
myFunc( Location loc )
```

```
        {
loc.x = 50;
loc.y = 100;
            Console.WriteLine( "In MyFunc loc: {0}", loc );
        }
        static void Main()
        {
            Location loc1 = new Location( 200, 300 );
            Console.WriteLine( "Loc1 location: {0}", loc1 );
Tester t = new Tester();
            t.myFunc( loc1 );
            Console.WriteLine("Loc1 location: {0}", loc1 );
        }
    }
}
```

Enumeratori

Nabrajanja (engl. enumerations) su moćna alternative konstantama. Predstavljaju vrednosni tip podataka koji predstavlja skup imenovanih konstanti.

Deklaracija i upotreba enum-a

```
enum Temperature
{
        ApsolutnaNula = -273,
        TackaSmrzavanja = 0,
        TackaKlucanja = 100,
};
```

Ovom enumeracijom postižemo bolju logičku vezu u kodu, nego da smo recimo kompletnu funkcionalnost uradili bez enumeratora. Ista stvar bi mogla da se realizuje na sledeći način:

```
const int ApsolutnaNula = -273;
const int tackaSmryavanja = 0;
const int tackaKljucanja = 100;
```

ali bismo ovakvom upotrebom u kasnijem delu povećali složenost održavanja, a smanjili logičku povezanost.

Referenciranje i konverzija

Pakovanje (boxing)

Pakovanje je mehanizam koji od vrednosnog podatka pravi referentni objekat, odnosno pravi se primerak objekta na hipu u koji se kopira vrednosni podatak.

Primer –pakovanje int promenljive:

```
int i=10; object o=i;
System.Console.WriteLine("i="+i+", o="+o);
```

Primer –pakovanje long literala:

```
object longObj = 1000L;
```

Strukture se mogu konvertovati u tipove interfejsa koje implementiraju Primer –pakovanje struct podatka S koji implementira interfejs I:

`S s=new S(); I i=s;` Implicitno pakovanje:

prilikom dodele vrednosti (kao u gornjim primerima) prilikom pozivanja metoda strukture

prilikom prosledivanja vrednosnog argumenta gde se očekuje referentni objekat

Raspakivanje

Obrnut proces od pakovanja - od objekta koji sadrži prethodno spakovanu vrednost se pravi podatak vrednosnog tipa.

Nije moguce raspakivanje bilo kog objekta (onog koji ne sadrži spakovanu vrednost)

Primer –pakovanje i raspakivanje int promenljive: `int i=10; object o=i; int ii=(int)o;`

```
System.Console.WriteLine("i="+i+", o="+o+", ii="+ii);
```

Neophodna je eksplicitna konverzija (cast)

Izvršni sistem proverava tip konverzije koji mora da odgovara tipu spakovane vrednosti, a ako se ne koristi odgovarajuća konverzija ispaljuje se izuzetak `System.InvalidCastException.`

Napomena: Ako su performanse bitne treba izbegavati pakovanje/raspakivanje, jer troši vreme.

Parametri ref i out

Za prenos po referenci se koristi ključna reč `ref`, a za izlazni `out`

Ove ključne reči se koriste i u definiciji metoda i na mestu poziva

Prenos parametara po referenci omogućava bočne efekte metoda nad njima:

–ne pravi se kopija stvarnog argumenta već se izmene vrše nad njim

Izlazni parametri ne moraju da budu inicijalizovani pre prosleđivanja metodu

Ako se izlaznom parametru ne pridruži vrednost u metodu –greška

Primer:

```
public class C{ public static void
               M(out int x){x=5;}
               public static void Main(){int x; C.M(out
                     x);}
}
```

Konverzija

Da bismo realizovali konverziju bilo koje promenljive potrebno je da realizujemo proveru tipa. Ukoliko je tip kompatibilan, konverzija je moguća.

Šta je to provera tipa? Provera tipa je aktivnost koja obezbenuje primenu operatora na operande kompatibilnih tipova. (Operandi su promenljive, izrazi, parametri, a operatori: dodeljivanje, relacioni operatori, funkcije…).

Postoji dva tipa konverzije i to:

Eksplicitna konverzija
Eksplicitna konverzija iizmenu razlliičitih tipova mora biti specificirana!

Primer: Eksplicitna konverzija **long** u **int**:

```
int intValue = (int) longValue;
```

Implicitna konverzija
Implicitna konverzija je izmenu različitih tipova odrenena je definicijom jezika.

Primer: Implicitna (automatska) konverzija **int** u **long**:

```
int intValue = 123; long
longValue = intValue;
```

Nastavni čas 11 – Nizovi

Pojam nizova

Nizovi su indeksirane kolekcije objekata istog tipa. U programskom jeziku C# je bitna razlika od ostalih programskih jezika iz C porodice – nizovi u programskom jeziku C# predstavljaju objekte.

Upravo zbog ove činjenice nizovi imaju svoje metode i svojstva (`BinarySearch()`, `Clear()`, `Copy()`, `CreateInstance()`, `IndexOf()`, `LastIndexOf()`, `Reverse()`, `Sort()`, `IsFixedSize()`, `IsReadOnly()`, `IsSynchronized()`, `Length`, `Rank`, `SyncRoot`, `GetEnumerator()`, `GetLength()`, `GetLowerBound()`, `GetUpperBound()`, `Initialize()` i `setValue()`).

Vrste nizova: jednodimenzionalni, dvodimenzionalni, višedimenzionalni, ugnježdeni, pravougaoni.

Kada se deklaše neki niz u C#-u, automatski se kreira objekat u ugrađenoj klasi `System.Array` (korišćenje ove klase možete videti u naredbi Using na početku svoje aplikacije – `Using System.Array`).

Deklaracija nizova

Deklaracija nizova se realizuje prema sledećoj sintaksi:

```
tip [] ime_niza;
```

Konstrukcija nizova

Po izvršenoj deklaraciji niza, deklarisani niz I dalje ne postoji. Da bi se napravio taj niz, potrebno ga je konstruisati. Konstrukcija niza se realizuje na sledeći način:

```
Int [] mojNiz = new int[5];
```

Bitna stvar je napraviti diferencu između samog niza i elemenata niza. Konstruisanjem niza sa new Int[5] mi smo u stvari deklarisali i konstruisali niz od pet celobrojnih varijabli, a elementi tog niza (tih 5 varijabli) imaju podrazumevane vrednosti. Odnosno, konstruisali smo niz vrednosnog tipa sa 5 elemenata default vrednosti – 0.

Takođe, bitna razlika je i to da se elementi niza referentnog tipa ne inicijalizuju svojim podrazumevanim vrednostima, nego je podrazumevana vrednost null. Ovo je veoma važna činjenica jer će se prilikom pokušaja upotrebe elementa izazvati izuzetak!

Inicijalizacija nizova

Nakon izvršene deklaracije i konstrukcije niza, elementi tog novonastalog niza imaju svoje default vrednosti. Inicijalna default vrednost za brojeve, bez obzira da li su celi ili realni, je 0. Inicijalna realna vrednost za String-ove je prazan String. Inicijalna vrednost za objekte je null, što znači da objekat ne postoji. U mnogim situacijama je potrebno da, se definišu konkretne vrendosti niza na početku rada programa. Definisanje vrednosti elemenata niza se zove inicijalizacija, i to se može uraditi kao što je prikazano na primeru ispod. `int[] niz = new int[4]; niz[0]=2; niz[1]=4; niz[2]=6; niz[3]=8;`

Dvodimenzionalni I višedimenzionalni nizovi

Da bismo razumeli pojam dvodimenzionalnih, a pogotovo višedimenzionalnih nizova, najlakše je da nizove zamislimo kao redove mesta na koje se smeštaju vrednosti. Ako zamislimo nekoliko takvih redova, praktično, zamislili smo jedan dvodimenzionalni niz.
Analogno tome, nizovima se može dodati i treća dimenzija, i četvrta i tako dalje, ali je takvo stanje malo teže plastično prikazati.

C# podržava dva tipa višedimenzionalnih nizova i to: pravougaone i nazubljene.

Pravougaoni niz je niz sa dve ili više dimenzija. U klasičnom dvodimenzionalnom nizu, prva dimenzija je broj redova, a druga je broj kolona.

Deklaracija i konstrukcija niza bi bila realizovana na sledeći način:

```
Int [,] pravougliNiz = new int [2,3];
```

Popunjavanje ovog niza pomoću for petlje i prikaz bi se realizovali na sledeći način:

```
static void Main(string[] args)
        {
            int redovi = 3;
int kolone = 4;

            //deklaracija niza sa celobrojnim elementima
int[,] pravougaoniNiz = new int[redovi, kolone];

            //popunjavanje niza elementima koji predstavljaju zbir rednog broja
i kolone             for (int i = 0; i < redovi; i++) {                for (int
j = 0; j < kolone; j++) {                    pravougaoniNiz[i,j] = i + j;
                }
            }

            //prikaz sadrzaja niza
for (int i = 0; i < redovi; i++) {
for (int j = 0; j < kolone; j++) {
                    Console.WriteLine("PravougaoniNiz[" + i + "," + j +"]=" + i+j
+";\n");
                }
            }

            System.Console.Read();

        }
```

Rad sa elementima niza

U nastavku Vam je prikazan listing windows forme koja vrši prikaz realizacije unosa, sabiranja, određivanja sume parnih elemenata, broja negativnih elemenata niza, broja elemenata niza deljivih sa 5, određivanja maksimalnog elementa niza.

```
using System; using
System.Drawing; using
System.Collections; using
System.ComponentModel;
using System.Windows.Forms;
using System.Data;
namespace nizovi
    {          public class Niz :
System.Windows.Forms.Form
        {
        private System.Windows.Forms.Label lbn;         private
System.Windows.Forms.Label lbx;         private
System.Windows.Forms.TextBox tBx;         private
System.Windows.Forms.ListBox lBNiz;         private
System.Windows.Forms.Button btUnesi;         private
System.Windows.Forms.GroupBox groupBox1;         private
System.Windows.Forms.CheckBox cBBrojNegativnih;         private
System.Windows.Forms.CheckBox cBBrojDeljivihSa5;
private System.Windows.Forms.CheckBox cBMaxNiza;
private System.Windows.Forms.CheckBox cBSumaParnih;
private System.Windows.Forms.CheckBox cBSumaNiza;
private System.Windows.Forms.TextBox tBRezultat;
private System.Windows.Forms.Button btKraj;         private
System.Windows.Forms.CheckBox cBSrednjaVrednost;
         int n=0;         int i=0;         int[] x;
private System.Windows.Forms.Button btnIzracunaj;
public Niz()
        {
            InitializeComponent();
            x=new int[40]; //konstruktor za niz
        }
        static void Main()
        {
            Application.Run(new Niz());
        }
        private void btnUnesi_Click(object sender, System.EventArgs e)
        {
            lbx.Text="x[0]=";
tBx.Focus();
            x[i]=Convert.ToInt32(tBx.Text);
lBNiz.Items.Add("x["+i+"]="+x[i]);
i++;             lbx.Text="x["+i+"]=";
tBx.Text="";             tBx.Focus();
        }          private void btnKraj_Click(object sender,
System.EventArgs e)
        {              n=i; //odredjivanje broja elemenata u
nizu             btUnesi.Enabled=false; //kraj niza, nema vise
```

```
unosa             btIzracunaj.Enabled=true;
lbx.Text=""; //brisemo tekst x[...]=
            lbn.Text="Niz ima "+n.ToString()+" elemenata"; //n elemenata
        }
        void SumaNiza(int[]x,int n)
        {
int j,s;
for(j=0, s=0;
j<n;j++)
s+=x[j];
            tBRezultat.Text+="Suma elemenata niza je "+ s+"\r\n";
}
          void SrednjaVrednost(int[]x,int
n)
        {              int j;
float xsr;             for(j=0,
xsr=0; j<n;j++)
xsr+=x[j];             xsr/=n;
            tBRezultat.Text+="Srednja vrednost niza je "+ xsr+"\r\n";
        }          void
SumaParnih(int[]x,int n)
        {             int j,sp;
for(j=0, sp=0; j<n;j++)
if (x[j]%2==0) sp+=x[j];
            tBRezultat.Text+="Suma parnih elemenata niza je "+ sp+"\r\n";
        }          void
BrojNegativnih(int[]x,int n)
        {              int j,brneg;
for(j=0, brneg=0; j<n;j++)
if (x[j]<0) brneg++;
            tBRezultat.Text+="Broj negativnih elemenata je "+ brneg+"\r\n";
        }          void
BrojDeljSa5(int[]x,int n)
        {             int j,b5;
for(j=0, b5=0; j<n;j++)
if (x[j]%5==0) b5++;
            tBRezultat.Text+="Broj elemenata deljivih sa 5 je "+ b5+"\r\n";
        }          void
MaxNiza(int[]x,int n)
        {
            int j,max=-32000;
for(j=0; j<n;j++)             if
(x[j]>max) max=x[j];
            tBRezultat.Text+="Maksimalni element niza je "+ max+"\r\n";
        }          private void btIzracunaj_Click(object sender,
System.EventArgs e)
        {
tBRezultat.Text="";
            if (cBSumaNiza.Checked) SumaNiza(x,n);
if (cBSrednjaVrednost.Checked) SrednjaVrednost(x,n);
if (cBSumaParnih.Checked) SumaParnih(x,n);             if
(cBMaxNiza.Checked) MaxNiza(x,n);             if
(cBBrojDeljivihSa5.Checked) BrojDeljSa5(x,n);             if
(cBBrojNegativnih.Checked) BrojNegativnih(x,n);         }
```

```
    }
}
```

Nastavni čas 12 – Delegati, događaji i kolekcije

Pojam delegata

Delegat je pojam uveden sa C# progarmki. Delegati se koriste za prenos metoda kao argumenata drugim metodama. Iako delegati prestavljaju novu vrstu objekata, delegati se radi boljeg razumevanja mogu predstaviti kao pokazivači na funkcije, koje su implementirane u programskom C uz bitnu napomenu da su delegati tipski bezbedni (type-safe function pointer or a callback).

Kod svih objekata u programskom jeziku C# pri korišćenju neke klase/delegata prvo morate da deklarišete odgovarajući objekat koji je tipa te klase/delegata, a zatim da napravite primerak tog objekta. Deklaracijom delegata opisuje se vrsta metode koju delegat predstavlja.

Primer deklaracije delegata:

```
delegate void primerDelegat(int broj1, int broj2);
```

Prethodni primer opisuje metod koji vraća void, a kao parametre ima dve celobrojne vrednosti tipa int.

Analogno ovom primeru, možemo imati i primer u kome delegat vraća neku vrednost.

```
delegate int primerDelegat(int broj1, int broj2);
```

U ovom primeru data je deklaracija delegata koji vraća celobrojnu vrednost, a za ulazne argumente ima dve celobrojne vrednosti. Pretpostavimo da negde uvašoj klasi imate definisane dve funkcije koje odgovaraju delegatu, tj.da vraćaju int i da imaju dva argumenta tipa int. Da pojednostavimo izlaganje: u klasi imate dve funkcije koje imaju isti potpis.

Primer:

```
        int sabiranje(int broj1, int broj2 )
        {
            return broj1 + broj2;
        }          int oduzimanje(int
broj1, int broj2)
        {
            return broj1 - broj2;
}
```

Kada pravimo analogiju sa objektima ove dve funkcije odgovaraju klasama za tipične objekte. One definišu ponašanje delegata koje tek treba da kreiramo. Kreiranje delegata vrši se na istinačin kao i kod svih drugih objekata, pomoću operatora new.

Primer:

```
        mojDelegat zbir = new mojDelegat(sabiranje);
mojDelegat razlika = new mojDelegat(oduzimanje);
```

Posle ovoga imamo dva objekta zbir i razlika koji su po svojoj prirodi funkcije, a koje možemo koristiti na isti način kao i te funkcije kojima su oni opisani.

Primer upotrebe iz ovog primera je sledeći:

```
int rezultat = zbir (15, 7);
rezultat = razlika (15, 7);
```

Upotreba delegata

Delegati se najčesće koriste prilikom obrade dogadjaja, kod programiranja Windows formi, ali naravno, nije njihova primena samo u tom delu. Mogu se koristiti uvek kada se unapred ne zna koja metoda treba da se pozove, te se u tom slučaju treba delegirati. Recimo, dogadjaji su, u principu, jedan tip delegata koji se koristi da se prosledi informacija da se na nekom objektu "nešto" dogodilo. To se mnogo koristi kod Windows formi kad treba npr. poslati informaciju da se nekom objektu forme dogodilo nešto.

Postoje dva tipa delegata-jednosmerni (single-cast) koji pozivaju samo jednu metodu i višesmerni (multi-cast) kojem se moze dodeliti više metoda i oni ne mogu da vrate vrednost.
Kada se deklariše klasa, mogu se napraviti i objekti delegata koji se zatim koriste za pamćenje i pozivanje metoda sa odgovarajućim potpisom metode.

Delegati se realizuju u C#-u kroz klasu System.Delegate, koja moze da sadrži i adresu instance klase kao i pokazivač na metodu.

Pojam događaja

Događaji ili Events predstavljaju promene stanja u programu koje, kada se dogode, pozivaju odgovarajuće metode koje su zadužene za obradu.

Upotreba događaja

Da biste dodali događaj selektujte glavnu formu i kliknite na ikonicu munje u Properties dijalogu.

Kada dodajemo događaj klik dugmadima, to radimo duplim klikom miša. Na taj način dodajemo događaj koji je najspecifičniji za izabranu kontrolu ili formu. Međutim, prozor Events u okviru Properties dijaloga dozvoljava da sami izaberemo događaj koji želimo.

Događaj koje se obično kreira za forme je Load koji se poziva prilikom startovanja programa ili pri prikazivanja forme.

Dodavanje metoda za obradu događaja vrši se:

1. Upisivanjem željenog imena pored događaja i pritiskom na taster ENTER i tada će se automatski kreirati željeni metod
2. Dvostrukim klikom miša na ime događaja tom prilikom se automatski kreira metod koji će se pozvati za obradu događaja

Kolekcije

Kolekcije su standardne strukture podataka koje dopunjuju nizove. Nizovi su jedina kolekcija koja je ugrađena u C# jezik. Sve ostale kolekcije su ugrađene u prostor imena System.Collections.

Prostor imena System.Collections

Prostor imena System.Collections sadrži klase i interfejse koje definišu različite kolekcije objekata. Od klasa treba pomenuti ArrayList, Queue, Stack, Hashtable. Od interfejsa tu su ICollection, IEnumerable, IEnumerator, IDictionaryEnumerator, IList itd.

Interfejsi

Interfejsi u prostoru SystemCollection koji sun am trenutno zanimljivi su:

ICollection definiše veličinu i enumeratore za sve kolekcije.

IComparer sadrži metode za komparaciju dva objekta.

IDictionary predstavlja kolekciju parova (ključ,vrednost).

IDictionaryEnumerator - enumeracija kroz elemente rečnika.

IEnumerable sadrži enumerator koji omogućava jednostavnu iteraciju kroz kolekciju.

IEnumerator - jednostavna iteracija kroz kolekciju.

IList interfejs predstavlja kolekciju objekata kojima se može pristupati preko indeksa.

Klasa ArrayList

Elementima liste pristupa se preko indeksa kao i kod niza. Za razliku od niza nije neophodno unapred poznavati broj elemenata niza. Najčešće korišćene metode ove klase su: Metoda Add(object) dodaje objekat na kraj liste. Metoda Clear() briše sve elemente iz liste. Metoda Insert(pozicija, vrednost) ubacuje objekat vrednost na pozicuju pozicija. Metoda RemoveAt(index) briše elemenat sa indeksom index iz liste. GetEnumerator() metoda vraća iterator koji se koristi za iteraciju(prolaz) kroz elemente liste. Metoda Sort sortira elemente liste. Metoda Reverse() prikazuje elemente liste u inverznom redosledu. Metoda ToArray() kopira elemente liste u jednodimenzionalan niz.

Upotreba ArrayList kolekcije

U nastavku je prikazan način kreiranja ArrayList kolekcije, način dodavanja članova u listu kao i način brisanja članova sa kraja liste.

```
    public class ArrayListPrikaz
    {
        ArrayList al = new ArrayList();

        public void metod()
        {
al.Add(7);
al.Add("Bosko");
al.Add(12.123);
al.RemoveAt(0);
            IEnumerator mojEnumerator = al.GetEnumerator();
while (mojEnumerator.MoveNext())
            {
                Console.WriteLine(mojEnumerator.Current.ToString());
            }
```

```
        }
}
```

Iteraciju kroz listu realizuje metoda GetEnumerator koja vraća enumerator liste. While petlja služi za štampanje elemenata liste. Enumerator se na početku pozicionira ispred prvog elementa kolekcije. Prvim pozivom metode MoveNext on se pozicionira na prvi element liste, sledećim pozivom na drugi itd. Kada se stigne do kraja liste metoda MoveNext vraća false. Property Curent daje tekući element liste.

Da bi korišćenje kolekcija bilo moguće, učitavamo prostor imena Collections:

```
using System.Collections;
```

Red (Queue)

Redovi predstavljaju realizaciju FIFO (first-in, first-out) strukture podataka. Metoda Enqueue služi za dodavanje elemenata u red. Ulazni parametar ove metode je tipa object tj. može biti bilo koji tip podataka. Skidanje elemenata iz reda vrši se korišćenjem metode Dequeue koja nema ulazne parametre. Kada se pozove ova metoda iz reda se izbacuje najstariji član tj. član koji je prvi ubačen u red. Za jednostavan prolazak kroz red koristi se metoda GetEnumerator koja vraća enumerator reda.

U nastavku je ilustrovana upotreba reda. Najstariji član u redu je karakter ‘A' pa se pozivom metode Dequeue on izbacuje iz reda. Rezultat izvršavanja sekvence koda prikazan je ispod koda.

```
    public class QueuePrikaz
    {          public void
metod()
        {
            Queue mojRed = new Queue();
mojRed.Enqueue('E');
mojRed.Enqueue(12);
mojRed.Enqueue('F');
mojRed.Enqueue(17);
mojRed.Dequeue();
            IEnumerator mojEnumerator = mojRed.GetEnumerator();
while(mojEnumerator.MoveNext())
            {
                Console.WriteLine(mojEnumerator.Current.ToString() );
}

        }
    }
```

Stek (Stack)

Stack je LIFO (Last-in, first-out) struktura. Element koji se poslednji stavi na stek, prvi se skida sa steka. Metoda Push stavlja element na vrh steka. Metoda Pop skida

poslednje stavljeni element sa steka. Metoda GetEnumerator takođe daje enumerator steka koji služi za jednostavan prolaz kroz elemente steka.

Ovde je prikazan primer upotrebe stack strukture, dodavanje elemenata na stack, skidanje elemenata sa stack-a, prolazak kroz elemente steka.

```
    public class StackPrikaz
    {          public void
metod()
        {
            Stack mojStack = new Stack();
mojStack.Push("Hello");
mojStack.Push(1);
mojStack.Push("World");
mojStack.Push("!");
mojStack.Pop();
mojStack.Pop();
            IEnumerator mojEnumerator = mojStack.GetEnumerator();
while (mojEnumerator.MoveNext())
            {
                Console.WriteLine(mojEnumerator.Current);
            }



        }
}
```

Hash tabela

Hash tabela je struktura podataka dizajnirana za brzo pretraživanje. To se postiže dodeljivanjem ključa svakom objektu koji se čuva u tabeli. Stringu "Idi na sever" dodeljuje se ključ 'N'. Vrednosti koja odgovara ključu 'N' se pristupa korišćenjem izraza tabela['N'], gde je tabela instanca klase Hashtable. Metoda GetEnumerator() takođe vraća enumerator koji je tipa IDictionaryEnumerator. Pomoću ovog enumeratora se može prikazati ključ i njemu pridružena vrednost za sve elemente tabele.

```
    public class HashTablePrikaz
    {          public void
metod()
        {
            Hashtable tabela = new Hashtable();
tabela.Add('N', "Idi na sever");
tabela.Add('S', "Idi na jug");
tabela.Add('W', "Idi na zapad");
tabela.Add('E', "Idi na istok");
tabela.Add('Q', "Dovidjenja");
Console.WriteLine(tabela['N']);
            IDictionaryEnumerator mojEnumerator = tabela.GetEnumerator();
while (mojEnumerator.MoveNext())
            {
```

```
                Console.WriteLine(mojEnumerator.Key.ToString() + "----->" +
mojEnumerator.Value.ToString());
            }

        }
    }
```

Nastavni čas 13 – Izuzeci

Izuzeci

Greške su uvek moguće, mogu nastati u bilo kom trenutku izvršavanja programa. Ovom činjenicom stalno moramo da se rukovodimo.

Upravo zbog ove činjenice postoji glavna razlika između dobrog i lošeg softvera, odnosno, dobrog i lošeg programa. Dobar program je onaj koji je u stanju da identifikuje i obradi nastalu grešku.

Jedan od načina obrade grešaka je i obrada izuzetaka.

U .NET Frameworku postoji ugrađeni mehanizam koji obrađuje izuzetke. Njegove osnovne karakteristike su:

- služi za obradu grešaka
- pruža dovoljno informacija o nastaloj grešci
- omogućava da se za svaki tip greške kreira odgovarajući način obrade
- omogućava odvajanje logike programa od koda kojim se obrađuju greške
- bazira se na predstavljanju izuzetaka pomoću objekata.

Kod tradicionalne provere grešaka morali smo da pišemo kod sličan ovome:

```
int kodGreške = 0;
FileInfo izvor = new
FileInfo(“kod.cs”); if (kodGreške == -
1) goto greška; int dužina =
(int)izvor.Length; if (kodGreške == -2)
goto greška; char[] sadržaj = new
char[dužina]; if (kodGreške == -3) goto
greška; ...
greška: ...;
```

Vrednost promenljive kodGreške ukazuje na to da li je nastala greška, i ako jeste kog je tipa.

Kod ovakvog pristupa ispoljilo se dosta nedostataka, kao na primer:

- Logika i obrada grešaka se prepliću

Instrukcije za otkrivanje grešaka su veoma slične sve one testiraju istu promenljivu (kodGreške) korišćenjem naredbe if dosta ponovljenog koda

Kodovi grešaka sami po sebi nisu jasni šta označava vrednost -1 ?

vrednost koda greške, celobrojna vrednost, nema eksplicitno značenje odnosno ne opisuje grešku koju predstavlja

gledanje dokumentacije je zamorno i podložno greškama Kodovi grešaka se lako mogu prevideti

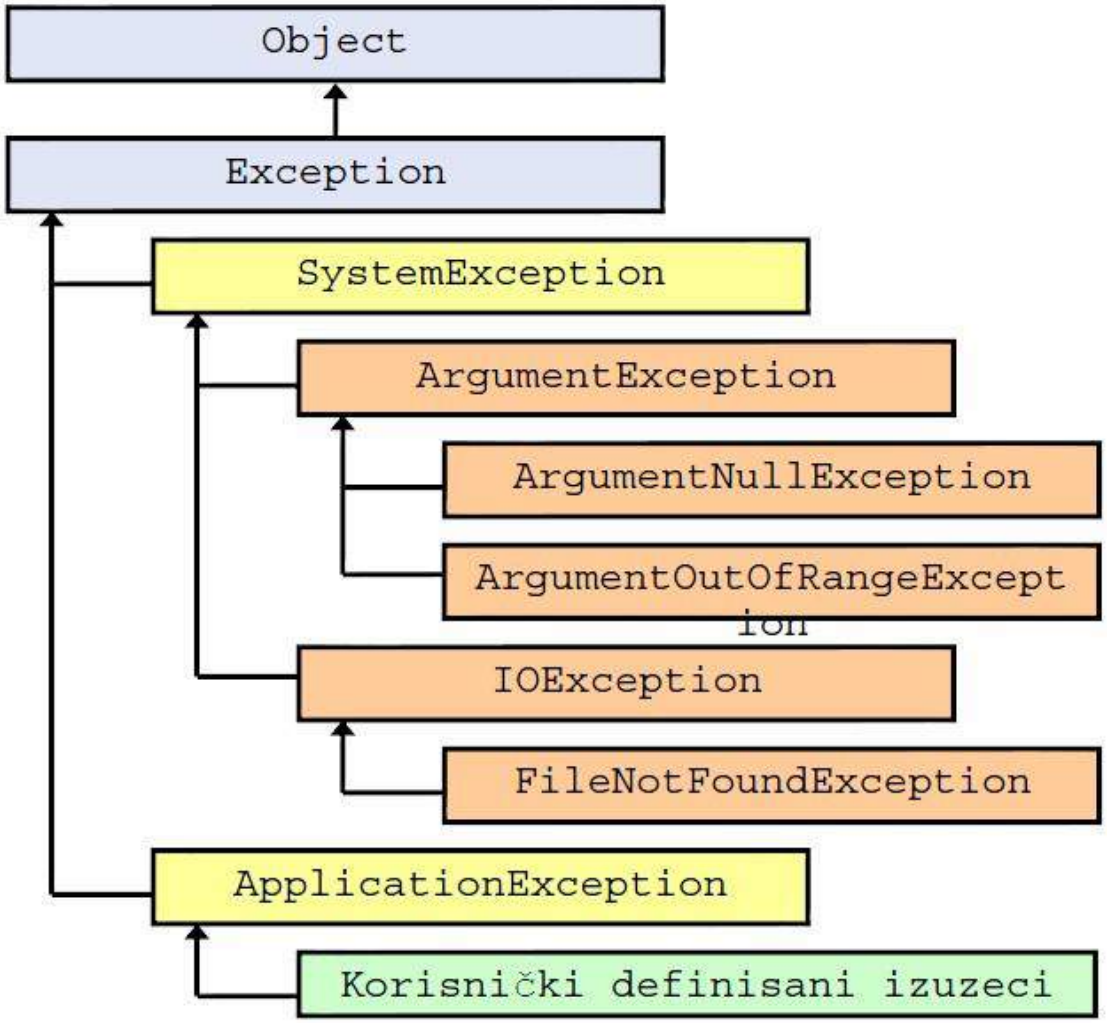
česta je situacija da se provera koda greške zanemari, a samim tim i obrada greške

Potreban je fleksibilniji mehanizam koji pruža dovoljno informacija o grešci

Da bi se prevazišao problem nedostatka informacija o nastalim greškama, .NET definiše niz različitih klasa izuzetaka.

Kako bismo mogli uopšte da pričamo dalje, potrebno je da definišemo šta je to izuzetak. Izuzetak je objekat koji se kreira ili podiže (throw) kada dođe do određene greške i sadrži informacije koje bi trebale omoguće identifikaciju greške.

Za potrebe obrade izuzetaka kreirana je kompletna hijerarhija klasa koje se bave isključivo izuzecima.



Osnovne karakteristike ovih klasa su da

svaka klasa može da se nađe unutar posebne izvorne datoteke i nezavisna je od drugih klasa i svaka klasa može da sadrži i njoj svojstvene podatke (npr. FileNotFoundException klasa mogla bi da sadrži ime datoteke koja nije pronađena.

Ove klase obezbeđuju sledeće prednosti:

Poruke o greškama se više ne predstavljaju brojevima, umesto njih koriste se odgovarajuće klase izuzetaka (npr. OutOfMemoryException klasa umesto -3)
Generišu se deskriptivne poruke o greškama. Svaka klasa se odnosi na određenu grešku i daje dovoljno jasan opis.

U hijerarhiji postoje dve osnovne klase koje nasleđuju System.Exception:

SystemException klasa - klasa iz koje se direktno ili indirektno izvode sve klase sistemskih izuzetaka
ApplicationException klasa - klasa iz koje se direktno ili indirektno izvode sve korisnički definisane klase izuzetaka

Specifičnost ove hijerarhije klasa izuzetaka se ogleda u tome što većina klasa nema nikakvu dodatnu funkcionalnost u odnosnu na svoje osnovne klase.

Kada se radi o obradi izuzetaka najčešći razlog nasleđivanja je da se ukaže na specifičnosti pojedinih grešaka i stoga nema potrebe da se dodaju nove metode ili pregaze nasleđene (iako nije redak slučaj dodavanja novih svojstava koja pružaju dodatne informacije o greškama).

Princip korišćenja ovih klasa na obradi izuzetaka je sledeći:

Kritični kod se deli na tri dela i to:

- try blok sadrži kod koji se odnosi na logiku programa u kome se mogu javiti ozbiljne greške.

- catch blok sadrži kod koji obrađuje različite tipove grešaka.

- finally blok sadrži kod kojim se oslobađaju resursi ili preduzimaju bilo kakve druge akcije koje bi trebalo da budu izvršene na kraju try ili catch blokova. Veoma je važno znati da se ovaj blok izvršava u svakom slučaju (bez obzira na to da li je podignut izuzetak)

Tok izvršavanja je sledeći:

Tok izvršavanja prelazi na try blok.
Ukoliko ne dođe do bilo kakve greške izvršavanje se normalno nastavlja sve do kraja ovog bloka kada se izvršavanje prenosi na finally blok. Međutim ukoliko dođe do greške unutar try bloka izvršavanje se prenosi catch blok.
U catch bloku se vrši obrada greške.
Na kraju catch bloka izvršavanje se automatski prenosi na finally blok. Finally blok se izvršava.

Načelno, realizacija hvatanja izuzetaka se realizuje prema sledećem:

```
try
{
    //logika programa
```

```
            }
catch
            {
                // obrada greške
            }
finally
            {
                // oslobadjanje resursa
            }
```

I pored ovoga, moguće je da nam finally blok nije potreban pa se on može izostaviti.

Takođe, moguće je da imamo potrebu da obradimo vise tipova izuzetaka u jednom problematičnom try bloku. U tom slučaju možemo kreirati situaciju sa višestrukim catch blokovima.

Najjednostavniji primer je prilikom parsiranja ulaza i deljenja tih brojeva.

```
            try
{
                int i = int.Parse(Console.ReadLine());
int j = int.Parse(Console.ReadLine());
int k = i / j;
            }
            catch (OverflowException ofe)
            {
                Console.WriteLine(ofe);
            }
            catch (DivideByZeroException dbze)
            {
                Console.WriteLine(dbze);
            }
```

Takođe, možemo imati i potrebu da definišemo izuzetak za koji ne znamo njegovu prirodu. U tom slučaju ćemo koristiti opšti catch blok. Primer korišćenja opšteg catch bloka:

```
            catch (System.Exception generalException)
            {
                //kod koji rukuje sa bilo kojim izuzetkom
}
```

Često korišćene klase izuzetaka:

ArithmeticException - osnovna klasa za izuzetke koji nastaju prilikom izvršavanja aritmetičkih operacija kao što su DivideByZeroException i OverflowException.

DivideByZeroException - podiže se prilikom pokušaja deljenja celobrojne vredenosti nulom.

ArrayTypeMismatchException - podiže se prilikom pokušaja da se u niz ubaci elemenat čiji tip nije kompatibilan sa tipom elemenata niza.

IndexOutOfRangeException - podiže se prilikom pokušaja pristupa nepostojećem elementu niza.

InvalidCastException - podiže se kada eksplicitna konverzija osnovnog tipa ili interfejsa u izvedeni tip ne može da se izvrši.

NullReferenceException - podiže se prilikom pokušaja korišćenja nepostojećeg objekta (vrednost reference je null).

OutOfMemoryException - podiže se prilikom neuspešnog pokušaja alociranja memorije (putem new).

OverflowException - podiže se kada aritmetička operacija izazove overflow (ukoliko je checked opcija uključena).

StackOverflowException - podiže se kada nema više mesta na steku usled prevelikog broja pozvanih metoda (npr. rekurzija).

FileNotFoundException - podiže se prilikom pokušaja da se pristupi datoteci koja ne postoji.

ArgumentException - podiže se kada vrednost nekog od prosleđenih parametara nije ispravna.

ArgumentNullException - podiže se kada se kao vrednost parametra prosledi null vrednost, a to nije dozvoljeno.

Nastavni čas 14 - Windows kontrole

Danas je potpuno nezamislivo da imamo neku apliakciju koja nema nikakvu interakciju sa korisnikom.

Da bismo kreirali svoju prvu aplikaciju koja će imati korisnički interfejs i koja će u startu moći nešto da prikaže na ekran, a da to nije konzolna aplikacija dovoljno je da kreiramo novi projekat prema opisu iz prve vežbe i sa malo rada postignemo željeni cilj. Upravo u tome i jesu čari vizuelnog programiranja – nije potrebno mnogo za inicijalne stvari u UI, ali daleko od toga da pravljanje funkcionalnog UI jednostavno.

Time se nemojte zavaravati!

Kada kreirate neku aplikaciju koja sadrži interakciju sa korisnikom moraćete da upotrebite objekte kojima obezbeđujete razmenu informacija izmedu aplikacije i korisnika. Ti objekti će prikazivati podatke korisniku ili će prenositi podatke od korisnika do vašeg programa. Oni se nazivaju kontrole.

Osobine Windows kontrola

Pre nego što se upustimo u dalje objašnjavanje kontrola, jedna važna napomena: pomen kontrole u ovoj vežbi odnosi se isključivo na kategoriju Windows kontrola.

Kontrole su prilično raznovrsne (ima mnogo različitih tipova) i one se postavljaju na formu po svojim svojstvima i po svojoj nameni. Istovremeno, iako su raznovrsne, postoje i mnoge sličnosti medu njima.

Zajednicko svojstvo za sve kontrole je da su po svojoj prirodi takode i prozori, odnosno imaju većinu sličnih svojstava. Njihov zadatak je da prihvataju određene događaje na interfejsu i omoguće dalji tok. Događaji su kao takvi, specifični samo za određenu kontrolu, ili su pak zajednički.

Posebno je bitno, kao što ste i u ranijim programerskim predmetima do sada imali, korišćenje specifične nomenklature prilikom imenovanja. Ime treba da bude opisno tj. Da što više opisuje ono što predstavlja. Ukoliko je za to potrebno više od jedne reči C# nomenklaturi takva imena počinju malim slovom a svaka nova reč velikim. Na primer: firstName, lastName, phoneNumber…

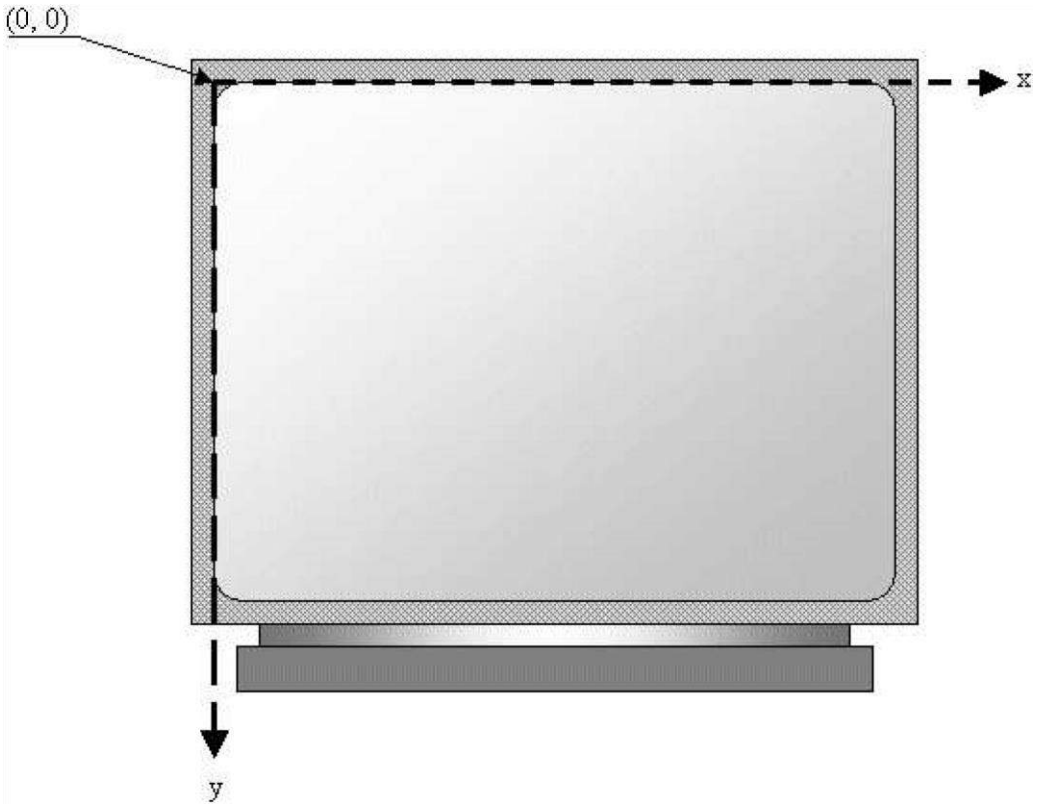
Razlozi za uvođenje nomenklature su jasni: održavanje koda, rad u timu, itd...

Relacija roditelj - dete

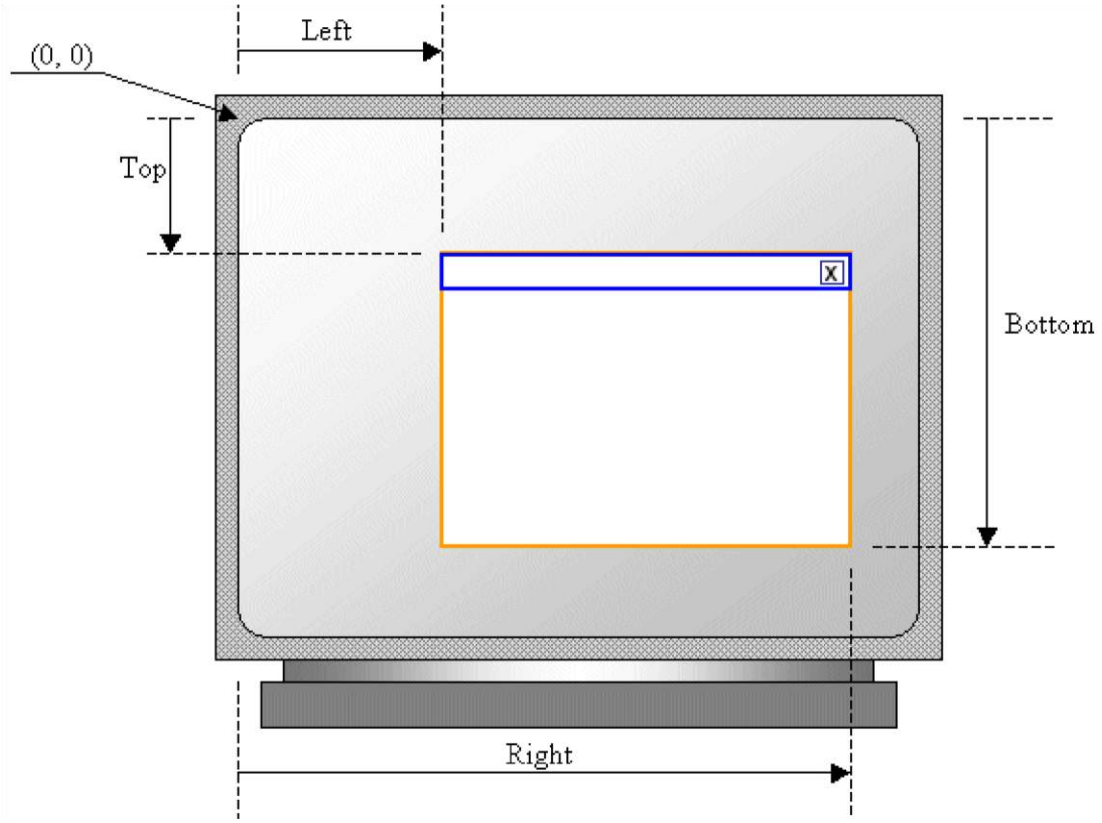
Postoji nekoliko razlicitih relacija izmedu prozora koji čine jednu ili više aplikacija. Za nas je za sada interesantan odnos je roditelj-dete tzv. parent-child relationship. Na primer, kontrola koju ste postavili na neku formu predstavlja prozor čiji roditelj je drugi prozor tj. Forma. Ovo je podrazumevana relacija kada se kontrole postavljaju i manifestuje se “lepljenjem” kontrole za tu formu. Pomeranjem forme pomeraju se sve kontrole na njoj, tj. sva deca prozori. Ista relacija postoji između formi koje pripadaju nakoj MDI aplikaciji.

Koordinatni sistem

Da bi se bolje razumelo svojstvo Location evo nekih dodatnih objašnjenja u vezi koordinatnog sistema koji se primenjuje kada je reč o postavljanju objekata na ekran i odnosima izmedu prozora u Windows operativnom sistemu.



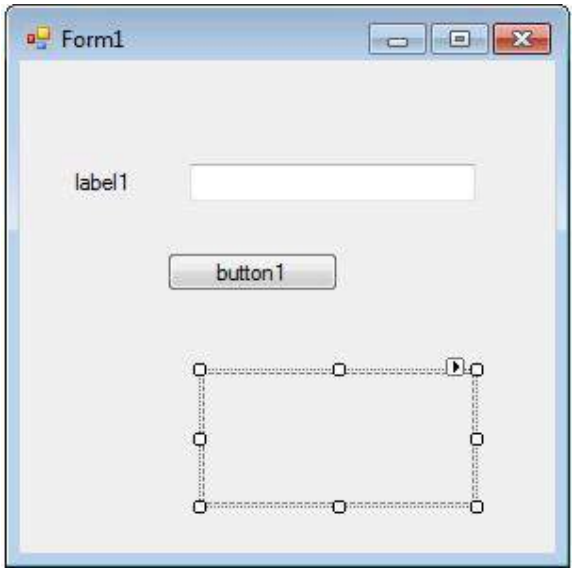
Pozicija nekog prozora čiji je roditelj desktop racuna se od gornjeg levog ugla ekrana, kao što je to prikazano na slici. Ose koordinatnog sistema orijentisane su tako da rastu od leve strane nadesno za x osu i od vrha nadole za y osu. Rastojanje od koordinatnog pocetka tacke do leve ivice je svojstvo “Left”. Adekvatna ostala svojstva zovu se “Right”, “Top”, “Bottom”.



Izbor i postavljanje kontrole na formu

Izbor kontrole se vrši sa toolbox palete na levoj strani ekrana. Kategorije na paleti toolbox obezbeduju nam brži izbor tražene kontrole.

Nakon selektovanja mišem, kontrola se uzima i otupšta (drag and drop) na tekući prozor, odnosno formu. Kontrola će biti pozicionirana na kursoru miša.



Pored ovog načina, postoji još nekoliko načina dodavanja kontrola na formu.

- Dvostruki klik na kontrolu - kontrola će biti automatski dodata formi u gornjem levom uglu;
- Postavljanje većeg broja istih kontrola na formu – pritisnut taster Ctrl i selekcija kontrole levim kliom miša.

Selekcija kontrole, podešavanje pozicije i velicine

U cilju promene nekih svojstava, kontrole i njene pozicije, kontrola se mora selektovati.Selekcija kontrole se realizuje levim klikom miša na odabranu kontrolu. Oko kontrole se pojavljuje pravougaonik sa 8 malih kvadratića koji olakšavaju promene dimenzija kontrole u odredenim pravcima.
U cilju selekcije više od jedne kontrole moraju biti pritisnuti tasteri Shift ili Ctrl. Tasterom Shift omogućena je selekcija kontrola od –do, a tasterom Ctrl je omogućeno selektovanje kontrola koje se odabiraju. Zatim se odabira jedna po jedna željena kontrola. Poslednja ima pravougaonik sa crnim kvadraticima oko nje. Drugi način da se selektuje skup kontrola je levim klikom miša na površinu forme i povlačenjem kursora se formira pravougaonik, a sve kontrole u njemu da postaju selektovane.
Pomeranje odredene kontrole vršite tako što kursorom selektujete tu kontrolu a zatim dok je levi klik miša pritisnut povlačite kontrolu preko forme. Pošto je pomerite na željenu poziciju otpustite taster miša i kontrola zauzima novu poziciju. Ako ste selektovali više od jedne kontrole na isti način možete pomerati i grupu kontrola. Drugi način je da selektujete kontrolu a onda koristeći “strelice” na tastaturi izvodite

pomeranje kontrole po formi. Korak pomeraja odgovara mreži na formi (grid). Ukoliko želite preciznije pomeranje koristite pritisnut taster Ctrl.

Zajednička svojstva Windows kontrola

Name

Svaka kontrola ima svojstvo Name. Smisao svojstva Name je da se preko njega omogući pristup. Ime mora postojati i ono se automatski dodaje pri kreiranju, tj. pri postavljanju neke kontrole/forme. Na primer, kada se neka aplikacija formira automatski se postavlja prva forma i njoj se dodeljuje ime Form1. Svaka sledeća forma dobija novo ime sa narednim brojem: Form1, Form2....
Isto pravilo važi i za kontrole koje se postavljaju na forme. One dobijaju imena po automatizmu koje im IDE okruženje dodeljuje a bazira se na tipu kontrole. Tako na primer tekst polja dobijaju imena text1, text2...

Location

Drugi primer uobičajenog svojstva svih kontrola je njihova lokacija (Location). Naravno ovo svojstvo je veoma bitno za vizuelni izgled forme, ali postoje takođe i kontrole koje se postavljaju na formu sa namerom da obezbede dodatnu funkcionalnost bez potrebe da budu vidljive. To je izuzetak kada pozicija kontrole nije od značaja. Ovo svojstvo je opisano koordinatama X i Y. Ovo je pozicija gornjeg levog ugla kontrole u odnosu na prozor na kome se kontrola nalazi. Obratite pažnju da je rec o vrednosnom tipu, jer su X i Y definisani kao celi brojevi) a celi brojevi se realizuju kao strukture.

Size

Kada je rec o grafičkim svojstvima kontrole, osim pozicije, treba spomenuti i njenu veličinu Size. Svi objekti su po prirodi pravougaonog oblika pa se velicina jednostavno opisuje. Svojstva koja opisuju velicinu su: širina (Width), odnosno visina (Height). Ukoliko se kontrola ne postavlja programski već u toku dizajna forme, promena veličine se obavlja jednostavnim korišćenjem miša, uz pomoć pravougaonika koji označava selektovanu kontrolu. Na pravougaoniku se mišem pritisne kvadratić zavisno od dimenzije koju želimo da promenimo, a zatim se povlačenjem miša menja dimenzija kontrole.
Drugi način da se promene parametri kontrole (ne samo veličina) je koristeci Properties prozor. Medu svojstvima kontrole lako se prepoznaje svojstvo Size ispred koga se nalazi znak +. Ovaj znak, tipično za Windows, omogućuje ekspandovanje stavke. Ispod ove stavke se nalaze posebno Width i Height.
Naravno, sva svojstva mogu se postavljati i programski. Bilo na samom startu pri kreiranju kontrola/formi, bilo u kodu.
Na primer:

```
static void Main()
{
    Form1 Fm = new Form1();
    Fm.Size = new System.Drawing.Size(800, 600);
    Application.Run(Fm);
}
```

Bounds

Pozicija zajedno sa veličinom smeštene su u jednom svojstvu kontrole koje se naziva Bounds. Ovo svojstvo daje pravougaonik kontrole/forme. U kodu ovim svojstvom lako se vrši repozicioniranje i promena veličine kontrole.

Text

Veoma korisno svojstvo svake kontrole/forme je Text. Naravno da postoji kod kontrola koje mogu prikazati tekst. Na primer, kod kontrole labela Text svojstvo čuva tekst koji kontrola prikazuje. Kod kontrole dugme predstavlja tekst koji se prikazuje na dugmetu. Dalje, kod forme predstavlja tekstualni sadržaj u naslovnoj liniji itd, itd. Promena ovog svojstva izvodi se u toku dizajna koristeci Properties prozor i biranjem Text polja a zatim ukucavanjem sadržaja. Obratite pažnju da je podrazumevani tekst u Unicode formatu, tj. lako možete koristiti srpska ili neka druga slova. Naravno da se tekst može menjati i programski.
Na primer:

```
static void Main()
{
    Form1 Fm = new Form1();
    Fm.Text = "New Application";
    Fm.Size = new System.Drawing.Size(800, 600);
    Application.Run(Fm);
}
```

Visible

Ranije smo pomenuli svojstvo koje odreduje da li je kontrola vidljiva ili ne. Ovo svojstvo se naziva Visible a vrednosti su bulovog tipa (Boolean type) True/False. Ako kontrola nije vidljiva onda se uobicajeno kaže da je sakrivena - hidden. Ovo svojstvo takodje se može programski menjati jednostavnim dodeljivanjem vrednosti. Ukoliko kontrola nije vidljiva onda nije ni dostupna – logicno. Drugim recima ona je za korisnika sa stanovišta unosa i prikaza informacija potpuno bez funkcije. To ne znaci da su takve kontrole nepotrebne – naprotiv. U toku izvršavanja programa one se mogu uciniti vidljivim i dostupnim, ili nevidljivim i nedostupnim, u zavisnosti od toga kakva vrsta podataka se unosi i šta se sa kontrolom radi.

Enabled

Kada je kontrola vidljiva to ne znaci obavezno da je i dostupna korisniku. Posebno svojstvo definiše da li kontrola može primiti ulaz od korisnika cak i kada je vidljiva. Ovo svojstvo se naziva Enabled i takode je bulovog tipa. Ako je kontrola vidljiva, a Enabled svojstvo postavljeno na vrednost false, to znaci da korisnik može samo citati vrednosti te kontrole bez mogucnosti da ih menja. Podrazumevani izgled kontrole u tom slucaju je blago zasivljenje (dimmed). Ovo svojstvo se cesto programski, u samom kodu menja. Obicno kada unos nekih podataka povremeno (ili stalno) treba zabraniti, a prikaz podataka ostaviti.

Fokus kontrole i aktivan prozor

Pojam fokusa i aktivnog prozora bitni su za razumevanje relacija medu prozorima i za objašnjavanje nekih svojstava (a kasnije i dogadaja). Za kontrolu se kaže da je u fokusu ako kontrola može da primi ulaz sa tastature. Takvo stanje poseduju kontrole, ali na primer i stavke u listi. Ono se može posebno graficki oznaciti što se u podrazumevanom vizuelnom podešavanju i radi.

Pojam aktivnog prozora ne vezuje se za kontrolu koja pripada nekoj formi vec za samu formu. Kontrola koja ima fokus nalazi se na aktivnom prozoru. Samo jedan prozor je aktivan u jednom trenutku. Na gornjoj slici dat je prikaz jednog primera. Jasno je da je Form1 aktivni prozor a textBox1 u fokusu i da se nalazi na tom prozoru.

TabIndex i TabStop

Ovde cemo pomenuti još jedno veoma bitno svojstvo u programiranju na grafickim okruženjima (GUI). Ovo svojstvo nije vidljivo – bar ne na prvi pogled. Obicno ga nazivamo tab redosled ili u originalu tab ordering. Kretanje sa kontrole na kontrolu na jednoj formi koja je aktivna možete izvesti i preko tastature koristeci Tab taster. Kada se on pritisne fokus kontrole se prebacuje na sledecu. Upravo tab svojstvo odreduje redosled kontrola u ovom slucaju.

I ne samo to. Kada govorimo o tab redosledu i tab svojstvima treba znati jos nešto. Postoje kontrole koje ne mogu primiti fokus – na primer labela. Takode, kontrola koja može da primi fokus mora imati svojsto TabStop postavljeno na true, ako treba da primi fokus. U suprotnom ta kontrola ce biti preskocena dok pritiskate taster Tab. Na srecu, ovo svojstvo je za kontrole koje mogu primati fokus inicijalno postavljeno na true. Kontrole koje ne mogu primiti fokus (primer Labela) nemaju ovo svojstvo. Koja pozicija u tab redosledu pripada odredenoj kontroli definiše svojstvo TabIndex. Ukoliko tu poziciju želite da promenite treba da promenite vrednost ovog svojstva.

Postavljanje kontrole u fokus

Ranije smo objasnili pojam fokusa kontrole u Windows operativnim sistemima. Naglasimo još jednom da fokus oznacava da je kontrola spremna da primi ulazne podatke. Na primer, ako se neko dugme nalazi u fokusu to se prikazuje tackastim pravougaonikom oko teksta na tom dugmetu (obratite pažnju na oznaku za podrazumevano dugme jer se ona cesto meša sa fokusom). Ako se tekst kontrola nalazi u fokusu onda se u njoj nalazi kursor koji trepce (tada je ocigledno da kontrola prima ulaz sa tastature). Kada se neka stavka liste nalazi u fokusu tada je ona oivicena tackastim pravougaonikom. Programski fokus dajemo nekoj kontroli pozivom metode Focus(). Na primer:

```
bool Focus();
```

Nastavno pitanje 15 – Windows kontrole – primeri korišćenja

U prethodnom delu smo se upoznali sa osnovnim osobinama windows kontrola, te u ovom delu možemo videti njihovu konkretnu primenu.

Button, TextBox, Label, GroupBox i Panel

U ovom delu nam je osnovno da uradimo iščitavanje unetog teksta iz tekst boksa. Recimo, možemo ga ispisati na novi panel.

- Kreirajte formu kao na slici:

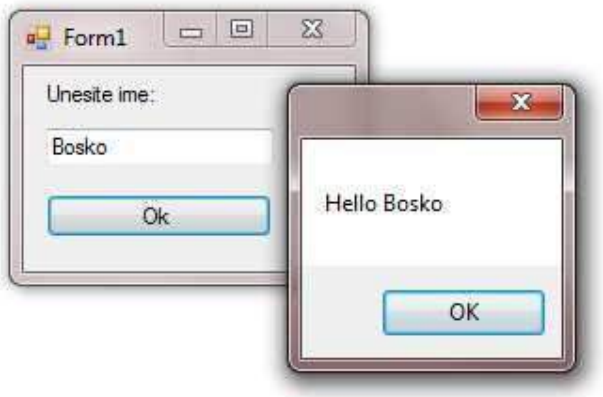
Da bismo postigli ovakvo ponašanje forme, potrebno je da uradimo sledeće:

Nakon kreiranja forme, odredimo akciju kojom ćemo realizovati našu zamisao. Kako je to akcija dugmeta click, uradimo dupli levi klik na dugme OK. Odmah nakon toga će se otvoriti prikay koda u kome ćemo moći da programskim kodom rešimo naš zadatak.

```
using System;
```

```
using
System.Collections.Generic;
using System.ComponentModel;
using System.Data; using
System.Drawing; using
System.Linq; using System.Text;
using System.Windows.Forms;
 namespace
WindowsFormsApplication286
{     public partial class Form1 :
Form
    {
        public Form1()
        {
            InitializeComponent();
        }
        private void button1_Click(object sender, EventArgs e)
        {
            MessageBox.Show("Hello " + textBox1.Text);
        }
    }
}
```

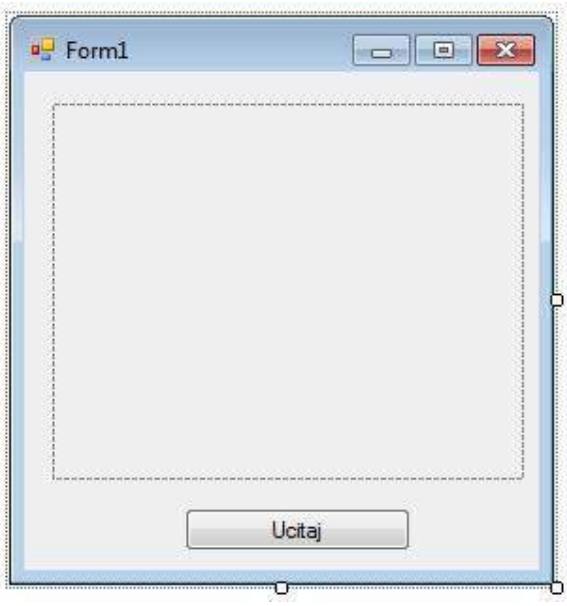
U okviru akcije dugmeta button1_Click unesite kod sa listinga. Time ćemo rešiti da nam se prikaže novi MessageBox sa prikazanim pozdravom i imenom koje smo uneli u tekst boksu.



PictureBox, OpenFileDialog

Pretpostavimo da želimo da odaberemo neku sliku iz našeg računara, da je učitamo I prikažemo u nekom prozoru. U tom slučaju biće nam potrebno nekoliko windows kontrola I naravno, nekoliko linija koda koje će to da omoguće.

Kreirajte formu kao na slici:



Vodite računa da je prostor koji je na slici u vežbi pictureBox.

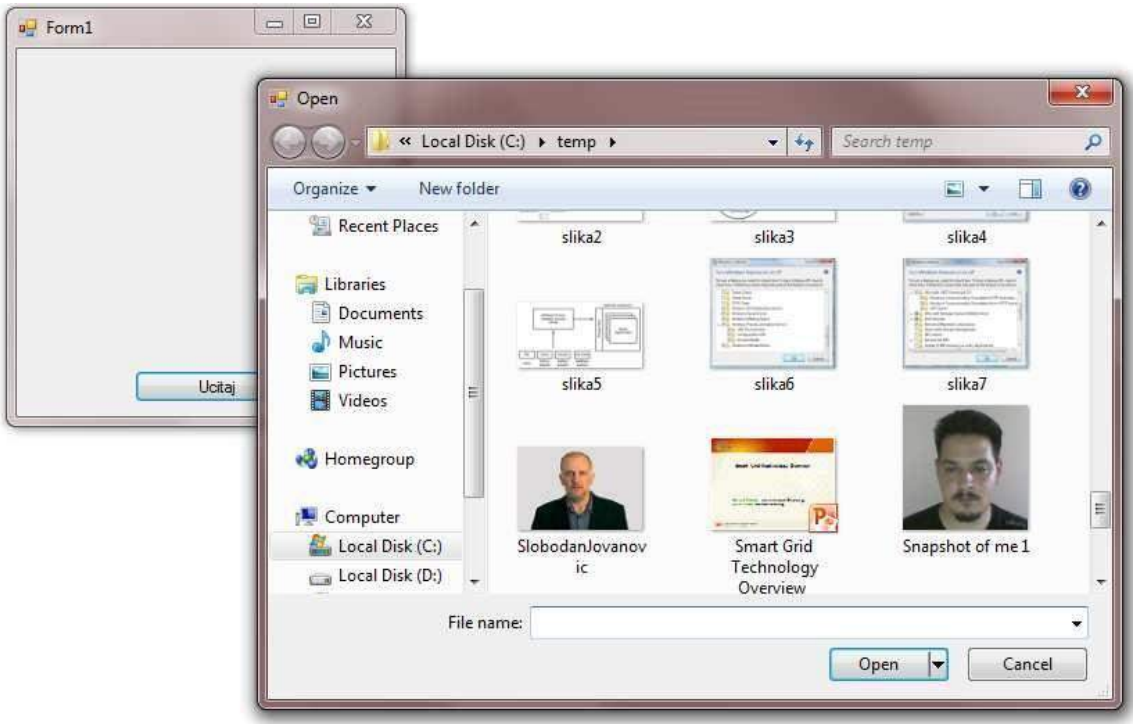
Nakon što ste ga kreirali, dodajte akciju dugmeta.
Za akciju dugmeta, posebno regulišite učitavanje fajla.

Pregled koda je dat ispod.

```
using System;
using
System.Collections.Generic;
using System.ComponentModel;
using System.Data; using
System.Drawing; using
System.Linq; using System.Text;
using System.Windows.Forms;
 namespace
WindowsFormsApplication286
{     public partial class Form1 :
Form
    {
        public Form1()
        {
            InitializeComponent();
        }
        private void button1_Click(object sender, EventArgs e)
        {
            OpenFileDialog of = new
OpenFileDialog();             DialogResult rez =
of.ShowDialog();             if (rez ==
DialogResult.OK)
            {
```

```
                Bitmap slika = (Bitmap)Bitmap.FromFile(of.FileName);
pictureBox1.Image = slika;
            }
        }
    }
}
```

Realizacijom ovog koda dobijamo učitavanje neke slike sa hard diska.



Odaberemo konkretnu sliku (u ovom primeru, prof. Dr. Jovanovića) i ona će biti učitana u pictureBox.

NAPOMENA: U cilju izbegavanja prevelikog rasplinjavanja u prikazu realizacije, u nastavku je dat samo potreban kod sa zadatkom. Vaš je zadatak da sve ove vežbe proradite i uverite se u način realizacije. Takvom realizacijom ćete i shvatiti upotrebu konkretne kontrole koja se obrađuje.

Text Box, button

Na osnovu unetog rednog broja dana u nedelji u jednom tekst boksu, ispisati naziv dana u drugom tekst boksu.

```
using System;
using
System.Collections.Generic;
using System.ComponentModel;
using System.Data; using
System.Drawing; using
System.Text; using
System.Windows.Forms; namespace
dani
{     public partial class Form1 :
Form
    {
        public Form1()
        {
            InitializeComponent();
tBRedniBrojDana.Text = "unesi";
        }
        private void btIspisi_Click(object sender, EventArgs e)
        {
int broj;
            if (int.TryParse(tBRedniBrojDana.Text, out broj))
            {
```

```
                tBNazivDana.Text = NazivDana(broj);
            }
else
            {
                MessageBox.Show("Redni broj dana nije dobro zadat!");
            }
        }          private string
NazivDana(int n)
        {
            string
rezultat;
switch (n)             {
                case 1: rezultat = "ponedeljak"; break;
case 2: rezultat = "utorak"; break;                 case 3:
rezultat = "sreda"; break;                 case 4: rezultat =
"cetvrtak"; break;                 case 5: rezultat = "petak";
break;                 case 6: rezultat = "subota"; break;
case 7: rezultat = "nedelja"; break;                 default:
rezultat = "nije dan u nedelji"; break;
            }
            return rezultat;
        }
    }
}
```

Timer, radio box, check box

Prikaz kontrola Timer, radio box I CheckBox kroz implementaciju brojanja.

```
using System; using
System.Drawing; using
System.Collections; using
System.ComponentModel;
using System.Windows.Forms;
using System.Data;
namespace Brojanje
{     public class Form1 :
System.Windows.Forms.Form
    {
        private System.Windows.Forms.Button btNapred;
private System.Windows.Forms.Button btNazad;
private System.Windows.Forms.Button btZaustavi;
private System.Windows.Forms.Button btPonisti;
private System.Windows.Forms.Timer timer1;         private
System.ComponentModel.IContainer components;         int
korak;
        private System.Windows.Forms.TextBox tbBroj;
public Form1()
        { InitializeComponent(); }
static void Main()
        { Application.Run(new Form1()); }
        private void btNapred_Click(object sender, System.EventArgs e)
        { timer1.Enabled = true; korak = 1; }
```

```
        private void btZaustavi_Click(object sender, System.EventArgs e)
        { timer1.Enabled = false; }
        private void btNazad_Click(object sender, System.EventArgs e)
        { timer1.Enabled = true; korak = -1; }
        private void btPonisti_Click(object sender, System.EventArgs e)
        { tbBroj.Text = "0"; timer1.Enabled = false; }
        private void timer1_Tick(object sender, System.EventArgs e)
        {             int broj =
Convert.ToInt32(tbBroj.Text);             broj =
broj + korak;             tbBroj.Text =
broj.ToString();
        }
    } }
```

CheckBox, label

Jednostavan primer za promenu fonta labele u zavisnosti od stanja check-box-a.

```
using System; using
System.Drawing; using
System.Collections; using
System.ComponentModel;
using System.Windows.Forms;
using System.Data;
namespace primer3a
{     public class CheckBoxTest :
System.Windows.Forms.Form
        {
        private System.Windows.Forms.CheckBox BoldCheckBox;
private System.Windows.Forms.CheckBox ItalicCheckBox;
private System.Windows.Forms.Label outputLabel;         private
System.ComponentModel.Container components = null;         public
CheckBoxTest()
        {
            InitializeComponent();
        }
        #region Windows Form Designer generated code
        #endregion

        [STAThread]
static void Main()
        {
            Application.Run(new CheckBoxTest());
        }
        private void BoldCheckBox_CheckedChanged(object sender, System.EventArgs e)
        {             outputLabel.Font=new
Font(outputLabel.Font.Name,
outputLabel.Font.Size,
            outputLabel.Font.Style ^ FontStyle.Bold);
        }
        private void ItalicCheckBox_CheckedChanged(object sender, System.EventArgs e)
```

```
        {
            outputLabel.Font=new Font(outputLabel.Font.Name,
outputLabel.Font.Size,
            outputLabel.Font.Style ^ FontStyle.Italic);
        }
    }
}
```

MessageBox, radio button

Jednostavan primer prikaza različitih MessageBox kontrola.

```
using System;
using
System.Collections.Generic;
using System.ComponentModel;
using System.Data; using
System.Drawing; using
System.Text; using
System.Windows.Forms; namespace
MessageBox1
{     public partial class Form1 :
Form
    {
        public Form1()
        {
            InitializeComponent();
        }         private void button1_Click(object sender,
EventArgs e)
        {
            MessageBox.Show("Zdravo!", "pozdrav", MessageBoxButtons.OK,
MessageBoxIcon.Exclamation);
        }
        private void button2_Click(object sender, EventArgs e)
{
            DialogResult rez;
            rez = MessageBox.Show("Da li zelite", "Brisanje",
MessageBoxButtons.YesNo, MessageBoxIcon.Question);
if (rez == DialogResult.Yes)
            {
                MessageBox.Show("Brisem!");
            }
else
            {
                MessageBox.Show("Nista nisam uradio!");
            }
        }
    } }
```

Radio Button, listBox

Jednostavan prikaz forme koja rukuje temperaturama pomoću radio button kontrole I list box kontrole.

```
using System; using
System.Drawing; using
System.Collections; using
System.ComponentModel;
using System.Windows.Forms;
using System.Data;
namespace temperature
{     public class temperature :
System.Windows.Forms.Form
    {
        int min, max, s = 0;
        private System.Windows.Forms.Label lTemp;
private System.Windows.Forms.TextBox tBTemperatura;
private System.Windows.Forms.Button btDodaj;
private System.Windows.Forms.ListBox lBTemperatura;
private System.Windows.Forms.GroupBox gBIzbor;
private System.Windows.Forms.RadioButton rBRaspon;
private System.Windows.Forms.RadioButton rBMin;
private System.Windows.Forms.RadioButton rBMax;
private System.Windows.Forms.RadioButton rBProsek;
private System.Windows.Forms.Label lIspis;         public
temperature()         { InitializeComponent(); }
static void Main()
        { Application.Run(new temperature()); }
        private void btDodaj_Click(object sender, System.EventArgs e)
        {
            lBTemperatura.Items.Add(tBTemperatura.Text);
            if (lBTemperatura.Items.Count == 1)//postavljanje min i max na prvu unetu
min = max = Convert.ToInt32(tBTemperatura.Text);
            s = s + Convert.ToInt32(tBTemperatura.Text); // dodavanje
temperature zbiru s             if (max < Convert.ToInt32(tBTemperatura.Text))
// korekcija min i max                 max =
Convert.ToInt32(tBTemperatura.Text);             else if (min >
Convert.ToInt32(tBTemperatura.Text))                 min =
Convert.ToInt32(tBTemperatura.Text);
            gBIzbor.Enabled = true; //moze se birati tek kad se unese prva t
lIspis.Text = ""; //priprema za unos nove t, sve se uncheck i brise
tBTemperatura.Text = ""; tBTemperatura.Focus();
            rBMax.Checked = rBMin.Checked = rBProsek.Checked = rBRaspon.Checked
= false;         }         private void rbProsek_CheckedChanged(object sender,
System.EventArgs e)
        {
            if (rBProsek.Checked) //zbir se deli brojem temp u listi
{ float p = (float)s / lBTemperatura.Items.Count; lIspis.Text =
p.ToString("0.00"); }
        }         private void rbMax_CheckedChanged(object sender,
System.EventArgs e)
        { if (rBMax.Checked) lIspis.Text = max.ToString(); }
```

```
    private void rBMin_CheckedChanged(object sender, System.EventArgs e)
    { if (rBMin.Checked) lIspis.Text = min.ToString(); }
    private void rbRaspon_CheckedChanged(object sender, System.EventArgs e)
    { if (rBRaspon.Checked) lIspis.Text = (max - min).ToString(); }
} }
```