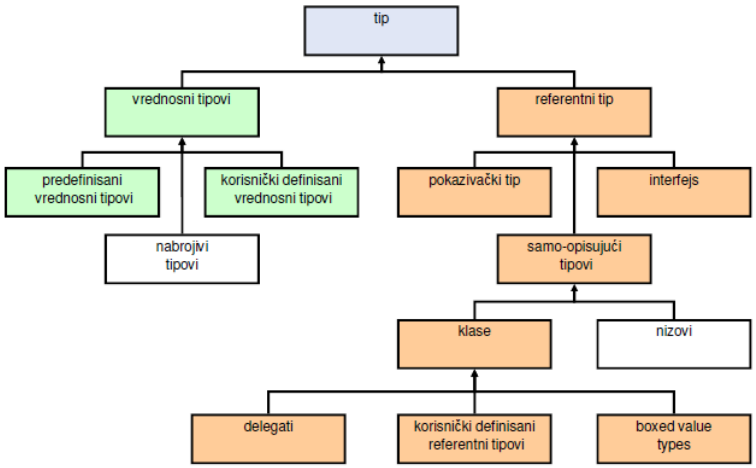
Tipovi podataka

Common Type System - CTS

- Common Type System (Sistem Opštih Tipova) je sastavni deo CLR-a.
- Definiše pravila za deklarisanje, korišćenje i upravljanje tipovima.
- Dati du svi predefinisani tipovi podataka koji su dostupni u IL-u.
- Podržava i vrednosne i referentne tipove.
- Od tipa podataka koje promenjiva treba da sadrži zavisi da li će ona biti vrednosnog ili referentnog tipa.
- Object je osnovna klasa koja predstavlja bilo koji tip, odnosno iz nje se izvode svi ostali tipovi.
- C# ima 15 predefinisanih tipova od kojih su 13 vrednosni, a 2 su referentni tipovi (string I object).

Hijerarhijska struktura CTS-a



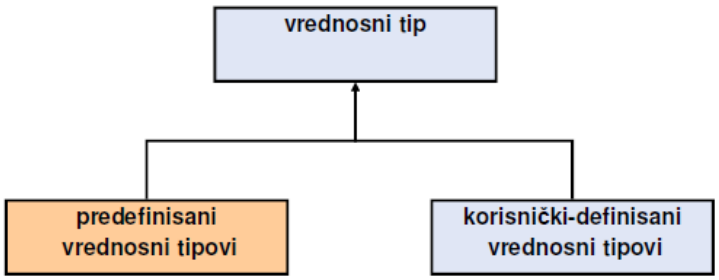
Vrednosni tipovi

- Promenljive vrednosnog tipa direktno sadrže podatke.
- Svaka promenljiva ima svoju kopiju podataka, tako da se ne može desiti da izvršavanje operacija nad jednom promenljivom utiče na drugu.
- Promenljive vrednosnog tipa se čuvaju na steku.

Promenljive vrednosnog tipa
Direktno sadrže podatke
Čuvaju se na steku.
Svaka ima svoju kopiju podataka - ne može se desiti da izvršavanje operacija nad jednom promenljivom utiče na drugu.

Predefinisani vrednosni tipovi

- Celobrojni tipovi
- Tipovi sa pokretnim zarezom
- Decimalni tip
- Logicki tip
- Znakovni tip



Predefinisani tipovi se navode pomoću ključnih reči ili odgovarajućih predefinisanih zapisa.

```
int System.Int32
float System.Single
bool System.Boolean
```

Naziv	CTS tip	Opis	Opseg	sufiks
sbyte	System.SByte	8-bitni označen	$-2^7 : 2^7-1$	
short	System.Int16	16-bitni označen	$-2^{15} : 2^{15}-1$	
int	System.Int32	32-bitni označen	$-2^{31} : 2^{31}-1$	
long	System.Int64	64-bitni označen	$-2^{63} : 2^{63}-1$	L
byte	System.Byte	8-bitni neoznačen	$0 : 2^8-1$	
ushort	System.UInt16	16-bitni neoznačen	$0 : 2^{16}-1$	
uint	System.UInt32	32-bitni neoznačen	$0 : 2^{32}-1$	U
ulong	System.UInt64	64-bitni neoznačen	$0 : 2^{64}-1$	U, L, UL

naziv:	float
CTS tip:	System.Single
opis:	32-bitni single-precision pokretni zarez
značajnih cifara:	7
približni opseg:	$\pm 1,5 \times 10^{-45} : \pm 3,4 \times 10^{38}$
sufiks:	F

naziv:	double
CTS tip:	System.Double
opis:	64-bitni double-precision pokretni zarez
značajnih cifara:	15/16
približni opseg:	$\pm 5,0 \times 10^{-324} : \pm 1,7 \times 10^{308}$
sufiks:	D

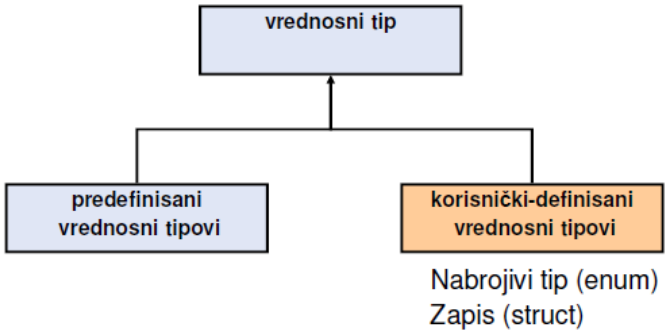
naziv:	decimal
CTS tip:	System.Decimal
opis:	128-bitni decimalni broj velike preciznosti
značajnih cifara:	28
približni opseg:	$\pm 1,0 \times 10^{-28} : \pm 7,9 \times 10^{28}$
sufiks:	M

naziv:	bool
CTS tip:	System.Boolean
vrednosti:	true, false

naziv:	char
CTS tip:	System.Char
vrednosti:	predstavlja jedan 16-bitni (Unicode) znak

Escape sekvenca	Znak	Escape sekvenca	Znak
\'	Apostrof	\f	form feed
\"	Znak navoda	\n	nova linija
\\	backslash	\r	carrige return
\0	null	\t	tab
\a	upozorenje	\v	vertikalni tab
\b	backspace		

Korisnički-definisani vrednosni tipovi



Nabrojivi tip

Za definisanje se koristi ključna reč enum iza koje se navodi naziv, a zatim imenovane konstante odvojene zarezima. Svi nabrojivi tipovi se implicitno izvode iz System.Enum tipa.

```
enum Boja //:int
{
Crvena, //=0
Zelena, //=1
}
```

• Svakom nabrojivom tipu se pridružuje numerički tip

- predstavlja tip vrednosti nabrojivog tipa
- može biti: byte, sbyte, short, ushort, int, uint, long ili ulong
- ukoliko nije eksplicitno naveden, podrazumeva se tip int

• Ukoliko nije drugačije navedeno, prva konstanta ima vrednost 0

- svaka sledeća ima vrednost za 1 veću od prethodne.

Nije dozvoljeno navođenje dve iste imenovane konstante.

```
enum Boja
{
  Crvena,
  Zelena,
  Crvena //greška
}
```

• Cirkularna zavisnost imenovanih konstanti nije dozvoljena.

```
enum Boja
{
  Crvena = Zelena, //greška
  Zelena
}
```

Nabrojivi tip

Više imenovanih konstanti može deliti istu vrednost.

```
enum Boja
 {
  Crvena,       //=0
  Zelena = 10,
  Plava,        //=11
  Maks = Crvena // Boja.Maks == Boja.Crvena == 0
}
...
Boja boja = Boja.Maks;
Console.WriteLine( (int)boja ); //0
```

Operatori koji mou da se primene nad vrednostima nabrojivog tipa – ==, !=, <, >, <=, >=, binarni +, binarni -, ^, &, |, ~, ++, --, sizeof

```
Boja boja = Boja.Crvena;
Console.WriteLine(boja++); // Zelena
```

Zapis

• Zapisi su slični klasama.
• Struktura podataka koja može da sadrži i podatke (koji mogu biti različitih tipova) i metode.
• Implicitno se izvode iz System.ValueType tipa.
• Koriste se za grupisanje konačnog broja promenljivih koje mogu biti različitih tipova.
• Za definisanje se koristi ključna reč struct iza koje se navodi naziv, a zatim polja i metode.
• Nije podržano nasleđivanje.

```
public struct Koordinata
{
public double x;
public double y;
}
```

Deklaracija i pristup poljima zapisa

Za pristup elementima zapisa koristi se kvalifikovano ime:
prvo se navodi naziv promenljive tipa zapis, zatim tačka (.), iza koje sledi naziv polja kome se želi pristupiti.

```
public struct Koordinata
{
public double x = 10; //greška
public double y;
}
```

• Deklaracijom se alocira prostor na staku za ceo zapis
• ne inicjalizuju se vrednosti polja.
• Prilikom deklaracije nije dozvoljeno navođenje inicijalnih vrednosti polja
• izuzetak su statička polja

```
Koordinata koordinata;
koordinata.x = 6.2;
koordinata.y = 12.6;
```

Konstruktori zapisa

• Prilikom deklaracije zapisa može se koristiti operator new

- ne alocira se memorijski prostor na heap-u, vec na staku !
- poziva odgovarajući konstruktor (u zavisnosti od prosleđenih parametara) kojim se inicijalizuju sva polja zapisa

```
Koordinata koordinata = new Koordinata();
koordinata.x = 6.2;
koordinata.y = 12.6;
```

• Automatski je podržan default konstruktor

- nema parametara.
- inicijalizuje sva polja zapisa na njihovu podrazumevanu vrednost.
- ne može se promeniti

Destruktor za zapis se ne može definisati.

Konstruktori zapisa se definišu na isti način kao i konstruktori klasa. Ograničenje nije dozvoljeno definisati konstruktor koji ne prima parametre.

```
public struct Koordinata
{
  public double x;
  public double y;

  public Koordinata(double a, double b)
 {
    x = a;
    y = b;
  }
}
...
Koordinata koordinata = new Koordinata(6.2, 12.6);
```

Referentni tipovi

Promenljive referentnog tipa sadrže reference na podatke, odnosno adresu na kojoj se mogu naći odgovarajući podaci. Podaci su smešteni u objektima. Pošto dve promenljive referentnog tipa mogu da referenciraju isti objekat, može se desiti da izvršavanje operacija nad jednom promenljivom utiče na objekat koji je referenciran drugom promenljivom.

Deklaracijom promenljive referentnog tipa alocira se memorija za referencu, ali ne i za instancu objekta datog tipa. Da bi se kreirao objekat koristi se ključna reč new. Za promenljive referentnog tipa postoji specijalna vrednost null.
null ukazuje na to da promenljiva ne sadrži referencu.

Promenljive referentnog tipa

Sadrže referencu na podatke, a ne same podatke. Podaci su smešteni u odvojenom delu memorije koji se naziva heap.
Deklarišu se na isti način kao i promenljive vrednosnog tipa.
public class Koordinata

```
{
  public double a;
  public double b;
}
...
Koordinata koord;
```

Deklaracijom promenljive referentnog tipa alocira se memorija za referencu, ali ne i za instancu objekta datog tipa.
Da bi se kreirao objekat koristi se ključna reč new.

Nakon što se referentnoj promenljivoj dodeli referenca na novi objekat, ona će nastaviti da pokazuje na taj objekat sve dok joj se ne dodeli referenca na novi objekat.

Neispravne reference

Referentnoj promenljivoj se može pristupiti promenljivima i metodama nekog objekta, samo ukoliko joj je inicijalizacijom dodeljena validna referenca.
Ovaj problem se može detektovati

- prilikom kompajliranja
- kompajler detektuje korišćenje neinicijalizovane reference

```
Koordinata c1;
c1.x = 6.2; // greška
```

prilikom izvršavanja pri pokušaju korišćenja referentne promenljive koja ima vrednost null sistem će podići NullReferenceException
ovo se može izbeći

```
try { c1.x = 6.2; }
catch (NullReferenceException)
{ Console.WriteLine( "c1 ima vrednost null"); }
```

Promenljive referentnog tipa

Pošto dve promenljive referentnog tipa mogu da referenciraju isti objekat, može se desiti da izvršavanje operacija nad jednom promenljivom utiče na objekat koji je referenciran drugom promenljivom.

System.Object

Bazna klasa za sve tipove.

- sve klase su direktno ili indirektno izvedene iz klase Object.
- metode koje nudi klasa Object, su prisutne u svim klasama bilo da su to sistemske ili korisnički definisane klase

Najčešće korišćene metode klase Object:

- ToString()
- Equals()
- GetType()
- Finalize()

Ove metode se u izvedenim klasama mogu reimplementirati, što je veoma čest slučaj.

Metoda >> ToString()
Vraća String kojim je predstavljen objekat. Puno kvalifikovano ime tipa.

```
public virtual string ToString();
Koordinata koordinata = new Koordinata();
Console.WriteLine(koordinata.ToString()); // Koordinata

Object obj = new Object();
Console.WriteLine(obj.ToString()); //System.Object
```

Ova metoda se može reimplementirati u izvedenim klasama. Kod predefinisanih vrednosnih tipova ova metoda vraća vrednost

```
int broj = 10;
Console.WriteLine(broj.ToString()); // 10
```

Metoda >> Equals()
Utvrđuje da li su dve instance objekata jednake. Proverava se jednakost reference.

```
public virtual bool Equals(object);
 public static bool Equals(object, object);
```

Ova metoda se može reimplementirati u izvedenim klasama. Poređenje instanci objekata prema vrednosti koju sadrže.

```
Koord koord = new Koord();
Koord koord1 = koord;

Console.WriteLine(koord.Equals(koord1));      // True
Console.WriteLine(koord.Equals(null));        // False
Console.WriteLine(object.Equals(null, null)); // True
```

Metoda >> GetType()
Vraća tip tekuće instance. Omogućava dobijanje informacija o tipu objekta u vreme izvršavanja.

```
public Type GetType();

public class OsnovnaKlasa {...}
public class IzvedenaKlasa: OsnovnaKlasa {...}
...
OsnovnaKlasa osnovna = new OsnovnaKlasa();
IzvedenaKlasa izvedena = new IzvedenaKlasa();

Console.WriteLine(osnovna.GetType());  // Osnovna
Console.WriteLine(izvedena.GetType()); // Izvedena

osnovna = izvedena;
Console.WriteLine(osnovna.GetType());  // Izvedena
```

Klasa String

- Ime string je skraćenica za System.String klasu.
- Za razliku od tipa Char kojim se prestavlja samo jedan znak
- string je sekvencijalni skup Unicode znakova, kojim se predstavlja tekst
- String je sekvenicijalni skup objekata tipa System.Char, kojim se predstavlja string.
- Vrednost String-a je sadržaj tog sekvencijalnog skupa i ona se ne može menjati nakon kreiranja,

- Ukoliko je potrebno izvršiti izmenu to se može izvršiti pozivanjem određene metode, kojom se ne menja sama instanca već se vraća nova instanca ovog tipa koja sadrži tu izmenu.
- indeks se predstavlja pozicija znaka, a ne sam znak u String-u.
- indeksi počinju od nula.

Svojstvo >> Chars

Vraća Unicode znak koji se nalazi na zadatoj poziciji (navedenoj indeksom) u toj instanci.
• ovo svojstvo je indekser String klase.
• parametar indeks polazi od nule.

public char this[int indeks] {get;}

indeks – pozicija nekog znaka u ovoj instanci.

```
Console.WriteLine(" Unesite string: ");
string mojString = Console.ReadLine();
for (int i = 0; i < mojString.Length; i ++)
Console.WriteLine("na poziciji {0} je {1}", i, mojString[i]);
```

Svojstvo >> Length

• Vraća broj znakova u toj instanci.
• vraća broj objekata tipa Char, a ne broj Unicode znakova (jedan Unicode znak može biti predstavljen sa više od jednog Char).

public int Length {get;}

```
string mojString = "ZDRAVO";
Console.WriteLine(mojString.Length); // 5
```

Metoda >> Insert()

• Ubacuje zadatu instancu tipa String na navedenu poziciju te instance.
• Vraća novu instancu tipa String koja je ekvivalentna toj instance ali sa ubačenom navedenom vrednošću.

public string Insert(int x string s);

• x – indeks pozicija od koje se ubacuje
• s – string koji treba ubaciti

Napomena: ukoliko je indeks pozicije x jednak dužini instance string s se nadovezuje na kraj instance.

```
string mojString = "C Community";
mojString = mojString.Insert(2, "# ");
Console.WriteLine(mojString); // C # Community
```

Metoda >> Copy()

Kreira se nova instanca tipa String kopiranjem navedene instance. Pravi duplikat navedene instance.

public static string Copy(string s);

s – string koji treba kopirati.

```
string mojString, noviString;
mojString = "ZDRAVO";
noviString = string.Copy(mojString);
```

```
Console.WriteLine(noviString); // ZDRAVO
```

Metoda >> Concat()

Spaja jednu ili više instanci tipa String, ili instanci Object-a čije su vrednosti reprezentovane String-om.

```
public static string Concat(object);
public static string Concat(params object[]);
public static string Concat(params string[]);
public static string Concat(object, object);
public static string Concat(string, string);
public static string Concat(object, object, object);
public static string Concat(string, string, string);
public static string Concat(string, string, string, string);
```

Kreira instancu tipa String koja predstavlja objekat.

public static string Concat(object x);

x – referenca tipa Object ili null
Napomena: umesto argumenta koji je null koristi se prazan string.

Spaja dve navedene instance tipa String.

public static string Concat(string x, string y);

x – prvi instanca tipa String
y – druga instanca tipa String

Napomena: umesto argumenta koji je null koristi se prazan string.

```
string ime = "Pera ";
string prezime = "PeriC";
Console.WriteLine(string.Concat(ime, prezime);// Pera PeriC
```

Poređenje vrednosnih i referentnih tipova

Vrednosni tipovi	Referentni tipovi
Podaci se čuvaju direktno	Podaci se čuvaju indirektno
Smeštaju se na stak	Objekat na heap / promenjiva na stack
Svaki ima svoju kopiju podatka	Dve promenjive mogu da referenciraju isti objekat
Operacije nad jednim ne mogu uticati na drugi	Mogu!!

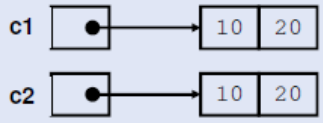
Poređenje vrednosti i poređenje referenci

Za poređenje vrednosnih tipova mogu se koristiti operatori == i !=.
Za poređenje referentnih tipova (izuzev stringova) ova dva operatora utvrđuju da li dve referentne promenljive pokazuju na isti objekat.

- ne poredi se sadržaj objekata na koje promenljive pokazuju

• za string tip == poredi vrednost stringova

```
Koordinata c1 = new Koordinata();
Koordinata c2 = new Koordinata();
c1.x =10;
c1.y =20;
c2.x =10;
c2.y =20;
if (c1 == c2)
   Console.WriteLine("isti");
else
   Console.WriteLine("različiti");    // rezultat je različiti
```



Ne koristiti ostale relacione operatore (>, <, >=, <=).

Konverzija tipova podataka

• Implicitna konverzija
• izvršava se automatski.
• uvek se uspešno izvršava i nikad ne dovodi do gubitka podataka.
• može dovesti do smanjenja preciznosti.

```
int broj1 = 6;
int broj2 = 5;
float broj = broj1 + broj2;
```

Eksplicitna konverzija - koristi se kod onih konverzija koje se ne mogu implicitno izvršiti, jer će kompajler javio grešku zbog toga što bi se izgubili podaci. koristi se cast izraz.

```
( tip ) promenljiva
```

• može dovesti do gubitka podataka.
• mogu se konvertovati samo numerički, znakovni i nabrojivi tipovi

```
double broj = 6.2;
int broj1 = (int) broj;
```

Implicitne numeričke konverzije

sbyte ⇨	short, int, long, float, double, decimal
byte ⇨	short, ushort, int, uint, long, ulong, float, double, decimal
short ⇨	int, long, float, double, decimal
ushort ⇨	int, uint, long, ulong, float, double, decimal
int ⇨	long, float, double, decimal
uint ⇨	long, ulong, float, double, decimal

Eksplicitne numeričke konverzije

sbyte ⇨	byte, ushort, uint, ulong, char
byte ⇨	sbyte, char
short ⇨	sbyte, byte, ushort, uint, ulong, char
ushort ⇨	sbyte, byte, short, or char
int ⇨	sbyte, byte, short, ushort, uint, ulong, char
uint ⇨	sbyte, byte, short, ushort, int, char
long ⇨	sbyte, byte, short, ushort, int, uint, ulong, char
ulong ⇨	sbyte, byte, short, ushort, int, uint, long, char
char ⇨	sbyte, byte, short
float ⇨	sbyte, byte, short, ushort, int, uint, long, ulong, char, decimal
double ⇨	sbyte, byte, short, ushort, int, uint, long, ulong, char, float, decimal
decimal ⇨	sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double

Promenljive

Promenljive

Promenljiva je imenovani memorijski prostor koji služi da se privremeno sačuvaju određeni podaci. Deklaracijom promenljive se alocira memorijski prostor.
Svaka promenljiva mora biti određenog tipa kojim se određuju dozvoljene vrednosti i operacije koje se mogu sprovesti nad njima. Veličina potrebnog memorijskog prostora.
Pre korišćenja promenljiva se mora deklarisati.

Ključne reči

Ključne reči se ne mogu koristiti kao nazivi promenljivih**.**

abstract	as	base	bool	break	byte
case	catch	char	checked	class	const
continue	decimal	default	delegate	do	double
else	enum	event	explicit	extern	false
finally	fixed	float	for	foreach	goto
if	implicit	in	int	interface	internal
is	lock	long	namespace	new	null
object	operator	out	override	params	private
protected	public	readonly	ref	return	sbyte
sealed	short	sizeof	stackalloc	static	string
struct	switch	this	throw	true	try
typeof	uint	ulong	unchecked	unsafe	ushort
using	virtual	void	volatile	while	

Deklaracija i inicijalizacija promenljivih

U opštem slučaju one se deklarišu tako što se specificira tip podataka iza koga se navodi ime promenljive.

Sintaksa:
```
tip_podataka identifikator;
```

Više promenljivih istog tipa se može deklarisati istovremeno.
```
int broj1, broj2;
```

Deklaracijom promenljive se samo alocira memorija.
Promenljiva mora biti inicijalizovana nekom početnom vrednošću pre nego što se nad njom može izvršiti bilo koja operacija.
Promenljive koje predstavljaju polja klase ili zapisa, ukoliko nisu eksplicitno inicijalizovane, imaju default vrednost od trenutka njihovog kreiranja.

Lokalne promenljive određene metode moraju biti eksplicitno inicijalizovane pre korišćenja, ali se to ne mora uraditi prilikom deklaracije.
Ukoliko se prilikom deklaracije koristi operator new, vrednost promenljive će biti inicijalizovana.

Dodeljivanje vrednosti promenljivoj
Dodeljivanje vrednosti promenljivoj koja je prethodno deklarisana, vrši se pomoću operatora dodele (=).

```
int broj;
broj = 20;
```

Promenljiva se može inicijalizovati prilikom njene deklaracije.
```
int broj = 20;
```
Operator dodele se može koristiti i za dodeljivanje složenih izraza.
```
broj = (broj * 2) + 40;
```

Opseg promenljive
• Deo koda u okviru koga se promenljivoj može pristupiti.
• Opseg važnosti lokalne promenljive je od trenutka njenog deklarisanja pa do kraja bloka u kome je deklarisana.
• Dve promenljive sa istim imenom ne mogu se deklarisati u istom bloku naredbi, i svim blokovima koji su ugnježđeni u taj blok.

```
int j=20;
for (int j=0; j<10; j++) //greška
{...}
} // promenjiva j izlazi iz opsega

for (int i=0; i<10; i++)
{...} // promenjiva i izlazi iz opsega
...
for (int i=9; i>0; i--)
{...} // promenjiva i izlazi iz opsega
```

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;
```

Ograničenje:
• kao konstanta se mogu navesti samo lokalna promenljiva ili polje klase.
Karakteristike:
• moraju biti inicijalizovane prilikom deklaracije, i kada im se jednom dodeli vrednost, ona se ne može promeniti.
• 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 njihove upotrebe: kod se lakše razume i modifikuje (promena se vrši samo na jednom mestu prilikom prepravljanja koda).

Operatori

Izrazi
Izrazi se grade pomoću:
- operanada
- literali, polja, lokalne promenljive i drugi izrazi i
- operatora
- ukazuju na operacije koje treba izvršiti nad operandima.
- Navođenjem operatora se ustvari poziva metoda kojim je implementiran taj operator.

Podržani operatori

aritmetički	+ - * / %
logički	& | ^ ~ && || !
spajanje stringova	+
inkrement, dekrement	++ --
šiftovanje bitova	<< >>
poređenja	== ! = <> >= <=
dodele	= += -= *= /= %= &= |= ^= >>= <<=
preusmeravanje i adresiranje	* -> & (samo u unsafe kodu) [ ]
kreiranje objekata	new
pristup članovima (za nizove,strukture)	.
indeksiranje(za nizove,indeksore)	[ ]
eksplicitna konverzija (cast)	()
uslovni ternarni operator	? :
kontrola overflow-a	checked unchecked
informacije o tipu	sizeof (samo u unsafe kodu) is typeof as

Operatori

Napomena:
operator dodele =
operator poređenja ==
Ternarni operator ?:
skraćeni oblik if... else naredbe
omogućava vrednovanje određenog izraza, u slučaju da je tačan vraća prvu vrednost, a u suprotnom drugu.

```
uslov ? true_value : false_value;
```

• uslov je logički izraz koji se ispituje.
• is operator
– omogućava proveru da li je izraz kompatibilan sa određenim tipom (ili nekim njegovim nasleđenim tipom) (izraz **is** tip)
//izraz mora biti referentnog tipa

Podržani operatori

as operator
Omogućava da se izvrše određeni eksplicitni tipovi konverzije (između kompatibilnih tipova).

```
izraz as tip; // izraz mora biti referentnog tipa
```

sizeof operator
Omogućava da se odredi veličina (u bajtovima) koja je neophodna da se smesti određeni vrednosni tip na stek. Može se koristiti samo u unsafe kodu.

```
sizeof (tip);
sizeof unarni_izraz;
```

• typeof operator
• za određeni tip vraća System.Type objekat.

```
typeof (tip);
```

Skraćeni zapis	Standardni zapis
x++, ++x	x = x + 1
x--, --x	x = x – 1
x += y	x = x + y
x -= y	x = x – y
x *= y	x = x * y
x /= y	x = x / y
x %= y	x = x % y
x >>= y	x = x >> y
x <<= y	x = x << y
x &= y	x = x & y
x |= y	x = x | y
x ^= y	x = x ^ y

Inkrementacija i dekrementacija
Često je potrebno povećati ili smanjiti određenu vrednost za 1.

```
broj = broj + 1; // broj +=1;
broj = broj -1; // broj -=1;
```

Zbog česte upotrebe postoji i kraći način zapisivanja.

```
broj++;
broj--;
```

++ operator se naziva inkrementni operator.
-- operator se naziva dekrementni operator.

Upotreba
Postoje 2 načina upotrebe ovih operatora:

1. ispred identifikatora (prefiks notacija)
2. povećava se vrednost promenljive pre upotrebe, tj. promenljiva se prvo inkrementira/dekrementira pa se zatim nova vrednost koristi u izrazu.

```
++broj
--broj
```

iza identifikatora (postfiks notacija)
originalna vrednost se nepromenjena koristi u izrazu, pa se tek onda inkrementira/dekrementira.

```
broj++
broj--
int broj=5;
if (++x == 6)
      Console.WriteLine (“ovo će se izvršiti”);
if (x++ == 7)
      Console.WriteLine (“ovo se neće izvršiti”);
```

Redosled navođenja operatora
Kada izraz sadrži više operatora njihov prioritet i redosled navođenja određuje redosled njihovog izvršavanja.
Kada se operand nađe između dva operatora istog prioriteta redosled njihovog izvršavanja je:
• sa leva na desno kod svih binarnih operatora
• sa desna na levo kod operatora dodele i uslovnog operatora (?:).

Na redosled izvršavanja se može uticati zagradama.

```
2 + 3 * 2 = 8 dok je (2 + 3) * 2 = 10
x = y = z se izvršava kao x = (y = z)
```

Grupa	Operatori
unarni	+ - ! ~ ++x --x casts
množenje/deljenje	* / %
sabiranje/oduzimanje	+ -
šiftovanje bitova	<< >>
relacioni	< > <= >= is as
poređenja	== ! =
Bitwise AND	&
Bitwise XOR	^
Bitwise OR	|
logičko AND	&&
logičko OR	||
ternarni operator	? :
dodele	= += -= /= *= %= &= |= ^=

Klase

Proceduralno programiranje

Nedostaci:

• svaki podprogram može da pristupi svakom podatku.
• promena u podacima može dovesti do neuspešnog izvršavanja podprograma.
• sa povećanjem veličine programa sve je teže vršiti izmene.

```
type
  tRacun = record
    brojRacuna: integer; stanje: real;
  end;
var
  racun1, racun2: tRacun;
  provizija: real;
  procedure Uplata(var racun: tRacun; iznos: real);
    begin
      racun.stanje := racun.stanje + (iznos*(1-provizija));
    end;
```

Objektno orijentisano programiranje

• 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.

Koncepti objektno orijentisanog programiranja:

- učaurenje
- nasleđivanje
- polimorfizam

OBJEKAT = PODACI + METODE

Apstrakcija

• Selektivno neznanje.
• Ignorisanje detalja
• Predstavlja zanemarivanje nebitnih detalja i usredsređivanje na bitne.

• Pristup “crne kutije” je da postoji mnogo objekata koje možemo da koristimo iako ne razumemo njihovu unutrašnju strukturu. Dva objekta mogu imati istu funkciju iako im je unutrašnjost potpuno različita.
• Postoje dve strane svakog objekta - ono što radi : što je obično poznato i način na koji radi : koji je obično nepoznat

Objekat

Objekat 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.
Objekat karakterišu:

- identitet - omogućava razlikovanje objekata među sobom (broj računa)
- ponašanje
  - dinamički aspekt objekta (uplata i isplata sa računa, ...)
  - definiše se metodama koje sadrži objekat
- stanje
  - statički aspekt objekta (broj računa, stanje na računu, vlasnik računa, ...)
  - definiše se podacima objekta i njegovim vezama sa drugim objektima u sistemu

Klasa

• Skup objekata koji imaju zajedničku strukturu i ponašanje.
• Klasa je generička definicija objekta.
• Klasa je struktura podataka koju treba posmatrati kao novi tip.
• Objekat je instanca klase.

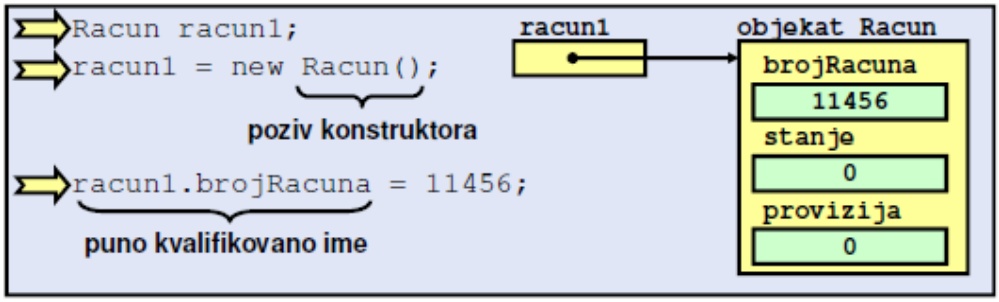
Definisanje klase

• Koristi se ključna reč class iza koje se navodi ime klase, a zatim se unutar vitičastih zagrada članovi klase.
• Članom klase se podrazumeva bilo koji podatak ili funkcija koji su definisani unutar klase.
  • pod funkcijom se podrazumeva bilo koji član koji sadrži kôd: metode, svojstva, konstuktori i preklopljeni operatori.
• Redosled navođenja članova klase nije značajan.

```
public class Racun
{
  public void Uplata(decimal iznos){...}          funkcije
  public void Isplata(decimal iznos){...}
  public void Prikazi(){...}

  private long brojRacuna;                          podaci
  private decimal stanje;
  private decimal provizija;
}
```

Instanciranje klase (kreiranje objekta)

• Klasa je referentni tip.
• deklaracijom se ne kreira instanca klase (objekat), već referenca.
• 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.



Konstruktori

• Specijalne metode koje služe za inicijalizaciju objekata nakon njihovog kreiranja.
• Konstruktor obezbeđuje da objekat ima dobro definisano početno stanje pre nego što se upotebi. Ako ne uspe inicijalizacija neće postojati objekat.
• Koristi se ime klase (metoda ima isto ime kao i klasa) iza koga slede zagrade.
• Nema povratnu vrednost, pa čak ni tipa void.
• Postoje dve vrste:
  • konstruktori instanci - oni inicijalizuju objekte
  • statički konstruktori - oni inicijalizuju klase

Podrazumevani (default) konstruktor

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

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

class Primer
{
    static void Main()
    {
         Datum danas = new Datum();   kreira se objekat i poziva konstruktor
        ...
    }
}
```

Osobine podrazumevanog konstruktora

• Ne prima parametre.
• Implicitno inicijalizuje sva nestatička polja na njihove podrazumevane vrednosti:
  • numerička polja (npr. 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

Reimplementiranje podrazumevanog konstruktora

Kada reimplementirati?
  • ako modifikator pristupa public nije odgovarajući
  • ako inicijalizacija polja na podrazumevane vrednosti nije odgovarajuća.

PREPORUKA: U konstruktoru bi trebalo da se rade samo jednostavne inicijalizacije

```
public class Datum
{
    private int godina, mesec, dan;

    public Datum()
    {
          godina = 2000;
          mesec = 1;
          dan = 1;
    }
}
```

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 Primer {
    static void Main()
    {
          Datum danas = new Datum(); //greška
    }
}
```

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)
    {
          godina = g;
          mesec = m;
    }
}
public class Primer
{
    static void Main()
    {
          Datum danas = new Datum(2010, 3);
          Console.WriteLine(danas.godina);   // ispisuje 2010
          Console.WriteLine(danas.dan);      // ispisuje 0
    }
}
```

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)

```
public class Datum
{
    public int godina, mesec, dan;

    public Datum ()
    {
    }

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

Konstruktori – Inicijalizatorske liste

- Može se desiti da preklopljeni konstruktori sadrže isti kôd.
- Inicijalizatorska lista omogućava da jedan konstruktor poziva drugi koji je deklarisan unutar iste klase.
- omogućava da se konstruktor implementira pozivanjem preklopljenog konstruktora.

SINTAKSA:

- navode se : iza kojih sledi ključna reč this, a u zagradi su navedeni parametri.
- Inicijalizatorske liste se mogu koristiti samo kod konstruktora

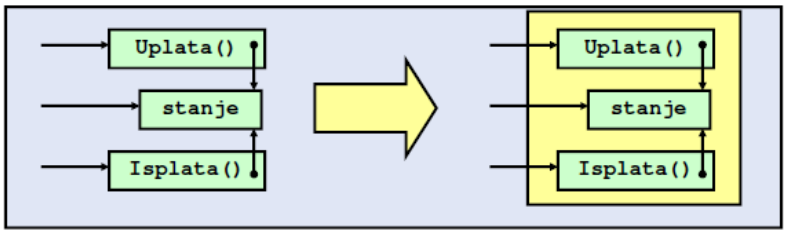
```
public class Datum
{
    public int godina, mesec, dan;
    public Datum (): this(2010, 3, 30){
    }
    public Datum (int g, int m, int d){
         godina = g;
         mesec = m;
         dan = d;
    }
}
```

Učaurenje

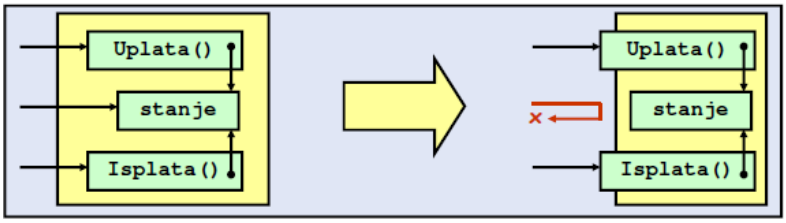
- Koncept objektno orijentisanog programiranja.
- Dodatna apstrakcija kojom se “sakrivaju” detalji implementacije objekta.
- Postoje dva bitna aspekta učaurenja:
- objedinjavanje podataka i funkcija u jedinstven entitet (klasa)
- kontrola mogućnosti pristupa članovima entiteta (modifikatori pristupa)

DIREKTAN PRISTUP PODACIMA JE I NEPOŽELJAN I NEPOTREBAN

- Objedinjavanje podataka i funkcija u jedinstven entitet
- ostvaruje se pomoću klasa.
- određuju se granice entiteta.



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




```
public class Racun
{
    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(100);
racun1.stanje = 555000; // nije obračunata provizija
```

PREPORUKA: podaci objekta treba da se nalaze u privatnom delu.

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

    private long brojRacuna;
    private decimal stanje;
    private decimal provizija;
}
...
Racun racun1 = new Racun();
racun1.Uplata(100);
racun1.stanje = 555000; // direktan pristup nije dozvoljen
```

Modifikator	Opis
public	Pristup bez ograničenja.
protected internal	Pristup je moguć iz sklopa u kome je definisan i izvedenih klasa klase u kojoj se nalazi, a koje su van tog sklopa.
internal	Pristup je moguć iz sklopa u kome je definisan.
protected	Pristup je moguć iz klase u kojoj je definisan i svih izvedenih klasa.
private	Pristup je moguć samo iz klase u kojoj je definisan.

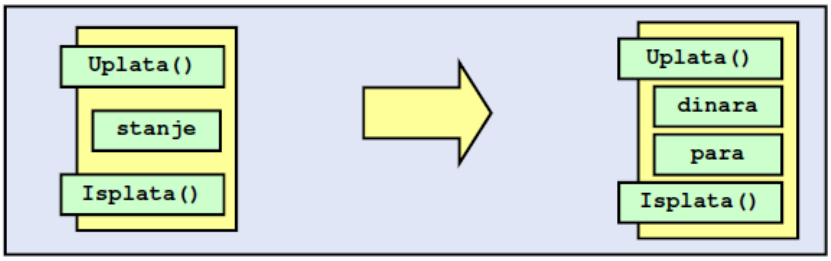
Od mesta deklaracije određenog člana zavisi i vrsta modifikatora pristupa koji se može koristiti. Ukoliko nije naveden, određuje se podrazumevani modifikator:
• za imenovani prostor nije dozvoljeno navođenje modifikatora, jer se podrazumeva da su javni tipovi koji se deklarišu unutar imenovanog prostora mogu biti public ili internal. Podrazumevani modifikator je internal.
• članovi klase mogu da imaju bilo koji od navedenih pet modifikatora. Podrazumevani je private.
• članovi zapisa mogu biti public, internal ili private. Podrazumevani 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 FON.PJ
{
    internal class Racun
    {
        ...
        public decimal stanje; //greška
    }
}
```

Zašto učaurivati?
Postoje dva razloga:
• omogućava se kontrola korišćenja
    • Objekat se može koristiti isključivo preko javnih metoda
• omogućava se smanjivanje 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)



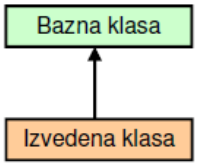

```
public class Racun {
    public decimal DajStanje()
    {
        return stanje;
    }
    public void Prikazi(){...}
    private decimal stanje;
}
public class Komitent {
    public void Prikazi()
    {
        Console.WriteLine( “Komitent ...” );
        Console.WriteLine( string.Format(“Stanje: {0}” racun.DajStanje() );
    }
    private Racun racun;
}
```

Promena ne utiče na korisnika klase.

```
public class Racun {
    public decimal DajStanje()
    {
        return (decimal)dinara + (decimal)para / 100;
    }
    public void Prikazi(){...}
    private long dinara;
    private long para;
}

public class Komitent {
    public void Prikazi()
    {
        Console.WriteLine( string.Format(“Stanje: {0}”, racun.DajStanje() ) );
    }
    private Racun racun;
}
```

Nasleđivanje

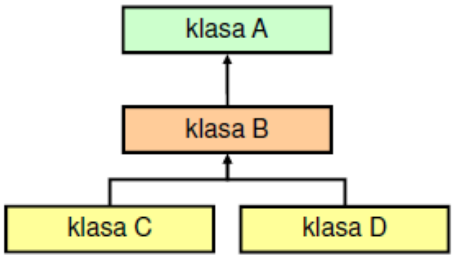
• Koncept objektno-orijentisanog programiranja.
• Omogućava da se na osnovu postojeće izvede nova klasa.
• Izvedena klasa nasleđuje sve članove bazne klase.
• Nasleđivanjem se ostvaruje veza (odnos) između objekata.
• is veza (“je tipa”)
• bazna (super, nadređena) klasa
• izvedena (podklasa, podređena) klasa koja nasleđuje baznu klasu.



Treba razlikovati

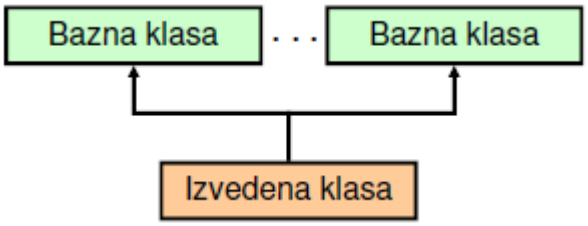
- nasleđivanje klasa
- nasleđivanje interfejsa
- klasa koja nasleđuje interfejs mora da implementira sve navedene funkcije

Nasleđivanje može biti direktno i indirektno.
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

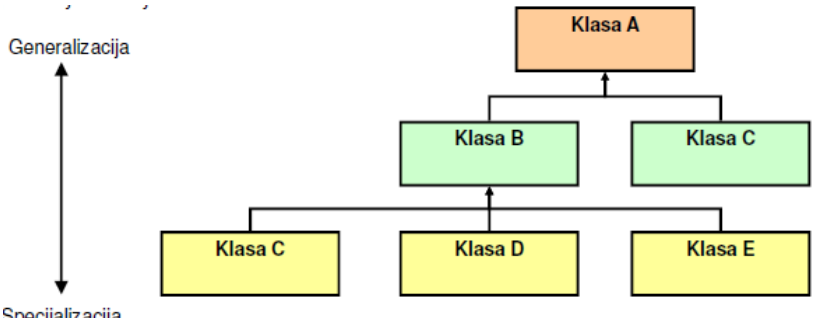


• Jednostruko nasleđivanje
  • Izvedena klasa direktno nasleđuje jednu baznu klasu.
• Višestruko nasleđivanje
  • Izvedena klasa direktno nasleđuje dve ili više baznih klasa

C# PODRŽAVA SAMO JEDNOSTRUKO NASLE_IVANJE



Hijerarhija nasleđivanja
• Svaka izvedena klasa može dalje biti bazna klasa.
• Grupa klasa koje su povezane nasleđivanjem formiraju strukturu koja se naziva hijerarhija klasa.
• klase na višim nivoima su opštije (koncept generalizacije), dok su one na nižim nivoima u hijerarhiji specifičnije (koncept specijalizacije).
• Dubina hijerarhije – broj nivoa nasleđivanja.
• preporuka je da broj nivoa ne bude veći od sedam



Sintaksa nasleđivanja

Ukoliko nije eksplicitno navedena bazna klasa podrazumeva se System.Object

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

• Izvedena klasa nasleđuje sve članove bazne klase osim konstruktora i destruktora.
• javni članovi bazne klase su implicitno javni članovi izvedene klase.

```
public class Racun
{
  public void Uplata(decimal iznos){...}
  public void Isplata(decimal iznos){...}
  private long brojRacuna;
  private decimal stanje;
}
public class TekuciRacun: Racun
{
}
...
TekuciRacun tekuci = new TekuciRacun();
Tekuci.Uplata(100); //pristup nasleđenoj metodi
```

Izvedenoj klasi se mogu dodati novi članovi.

```
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 član
}
```

Pristup članovima bazne klase

Nasleđivanje ne podrazumeva da će izvedena klasa imati pristup svim članovima bazne klase.

Privatni članovi bazne klase, iako nasleđeni, su dostupni isključivo članovima bazne klase.

```
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
  }
}
```

Pristup članovima bazne klase
Članovi bazne klase sa modifikatorom protected su jedino dostupni unutar bazne klase I direktno i indirektno izvedenim klasama. U ostalim slučajevima se ponašaju kao da su označeni sa "private"

PREPORUKA - sva polja klase treba da imaju modifikator pristupa private, a za svako od njih navesti svojstvo sa modifikatorom pristupa protected.

```
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.

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

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

```
...
private class Racun
{
...
}
public class TecuciRacun: Racun // greška
{
...
}
```

Poziv konstruktora bazne klase
- 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

```
public class BaznaKlasa
{
  public BaznaKlasa() {...}
}

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

inicijalizator konstruktora

Greška: poziva se nepostojeći konstruktor bazne klase

```
public class BaznaKlasa
{
  public BaznaKlasa(int x) {...}
}

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

```
public class IzvedenaKlasa: BaznaKlasa
{
    public IzvedenaKlasa(): base(10) {...} //OK
    public IzvedenaKlasa(int broj): base(broj) {...} //OK
}
```

Sakrivanje metoda

Ukoliko se u izvedenoj klasi navede identična metoda (metoda sa istim potpisom) kao i u baznoj klasi, prijaviće se upozorenje o preklapanju u nazivima metoda.

“FON.PJ.TekuciRacun.Prikazi()' hides inherited member 'FON.PJ.Racun.Prikazi()'. Use the new keyword if hiding was intended.”

```
namespace FON.PJ
{
    public class Racun
    {
        public void Prikazi(){
                Console.WriteLine(string.Format(“Racun: {0}”, brojRacuna));
        }
    }

    public class TekuciRacun: Racun
    {
        public void Prikazi(){
            Console.WriteLine(string,Format(“Tekuci racun: {0}”, brojRacuna));
        }
    }
}
```

Metode sa istim potpisom

Ova situacija se može razrešiti na sledeće načine:

1. promena naziva metode u izvedenoj klasi

2. navođenje ključne reči new ispred metode u izvedenoj klasi - nasleđena metoda se sakriva novom metodom u izvedenoj klasi
3. navođenje ključne reči virtual ispred metode u baznoj i ključne reči override u izvedenoj klasi - nasleđena metoda se reimplementira u izvedenoj klasi

Sakrivanje metoda

Promena naziva metode u izvedenoj klasi.

```
public class Racun
{
  public void Prikazi(){
    Console.WriteLine(string.Format("Racun: {0}", brojRacuna));
  }
}
public class TekuciRacun: Racun
{
  public void PrikaziTekuci(){
    Console.WriteLine(string.Format("Tekuci racun: {0}", brojRacuna));
  }
}
...
TekuciRacun tekuci = new TekuciRacun();
tekuci.Prikazi();       // Racun: 11456
tekuci.PrikaziTekuci(); // Tekuci racun: 11456
```

Da bi se sakrila nasleđena metoda bazne klase koristi se ključna reč new.

- uvodi se potpuno nova metoda kojom se zamenjuje nasleđena

Koja metoda će biti pozvana, metoda bazne ili izvedene klase?

- određuje se na osnovu tipa promenljive, a ne na osnovu tipa same instance

```
public class Racun {
  public void Prikazi() {
    Console.WriteLine("Racun: {0}", brojRacuna);
  }
}
public class TekuciRacun: Racun {
  new public void Prikazi() {
    Console.WriteLine("Tekuci racun: {0}", brojRacuna);
  }
}
Racun racun = new Racun();
racun.Prikazi();  // Racun; 11456
TekuciRacun tekuci = new TekuciRacun();
tekuci.Prikazi(); // Tekuci racun: 11456
```

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.
OsnovnaKlasa promenljiva = new IzvedenaKlasa();

KOMPATIBILNOST OBJEKTNIH TIPOVA

Promenljiva tipa bazna klasa može da referencira instance direktno ili indirektno izvedenih klasa

Polimorfizam se realizuje pomoću VIRTUELNIH METODA

Virtuelne metode

Za deklaraciju virtuelne metode se koristi ključna reč virtual.

Prilikom definisanja, virtuelne metode se moraju implementirati.

Privatne i statičke metode se ne mogu definisati kao virtuelne.

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

Reimplementiranje metoda

Proces implementacije virtuelne metode u izvedenoj klasi se naziva reimplementacija.

Koristi ključna reč override.

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

Kao i virtuelna metoda i reimplementirana metoda mora sadržati kod.

```
public class TekuciRacun: Racun
{
  public override void Prikazi(); //greška
}
```

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

```
public class Racun
{
public virtual void Prikazi() {...}
}
public class TekuciRacun: Racun
{
public override void Prikazi(string s) {...} //greška
}
```

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.

```
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; 11456
racun = new TekuciRacun();
racun.Prikazi(); // Tekuci racun: 11456
```

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

```
public class A
{
  public virtual void Prikazi() {...}
}
public class B: A
{
  public override void Prikazi() {...}
}
public class C: B
{
  public override void Prikazi() {...} // OK
  public override virtual void ...     // Greška
}
```

Rano i kasno povezivanje

- Rano povezivanje:
  - statičko povezivanje
  - poziv metode se razrešava u toku kompajliranja.
- Kasno povezivanje:
  - dinamičko povezivanje
  - poziv metode se razrešava u toku izvršavanja.
- Kasno povezivanje sa jedne strane povećava fleksibilnost, dok sa druge strane utiče na performanse.
  - potrebno je više vremena za povezivanje
  - potrebno je pronaći klasu čija je metoda pozvana.

Virtuelne metode

Metode nisu implicitno virtuelne iz sledećih razloga:

performanse

- potrebno je vreme da bi se odredilo (pronašlo) 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.

Sakrivanje virtuelnih metoda

• Virtuelne metode se takođe mogu sakriti
• Sakrivanje metoda utiče na polimorfizam
• Nova metoda može biti virtuelna, što znači da se u dalje izvedenim klasama ona može reimplementirati

```
public class A
{
  public virtual void Ispis() {...}
}
public class B: A
{
  public override void Ispis() {...}
}
public class C: B
{
  new public void Ispis() {...}
}
```

Sakrivanje vs. Reimplementiranje

```
public class OsnovnaKlasa
{
   public void Metoda() {Console.Write("A");}
}

public class IzvedenaKlasa: OsnovnaKlasa
{
   public new void Metoda() {Console.Write("B");}
}

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

Na ekranu se ispisuje?

• U vreme kompajliranja, na osnovu tipa promenljive, će se odrediti metoda koja se poziva.
• poziva se metoda osnovne klase ... ispisuje se A

```
public class OsnovnaKlasa
{
   public virtual void Metoda() {Console.Write("A");}
}

public class IzvedenaKlasa: OsnovnaKlasa
{
   public override void Metoda() {Console.Write("B");}
}

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

Generisaće se kod kojim će se proveriti tip instance koju referencira promenljiva, u ovom slučaju izvedena klasa
• poziva se reimplementirana metoda ... ispisuje se B

```
public class OsnovnaKlasa
{
   public virtual void Metoda() {Console.Write("A");}
}

public class IzvedenaKlasa: OsnovnaKlasa
{
}

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

Generisaće se kod kojim će se proveriti tip instance koju referencira promenljiva, u ovom slučaju izvedena klasa
• poziva se nasleđena metoda obzirom da nije reimplementirana ... ispisuje se A

Sakrivanje virtuelnih metoda

```
public class A {
  public virtual void Ispis() { Console.Write("A"); }
}
public class B: A {
  public override void Ispis() { Console.Write("B"); }
}
public class C: B {
  new public virtual void Ispis() { Console.Write("C"); }
}
public class D: C {
  public override void Ispis() { Console.Write("D"); }
}
...
D d = new D(); C c = d; B b = c; A a = b;
d.Ispis(); c.Ispis(); b.Ispis(); a.Ispis();
```

Na ekranu se ispisuje?
DDBB

Metode

Metode
• Prilikom dizajniranja većine aplikacija vrši se njihova podela na funkcionalne jedinice
• prednost:
  • jedinice se mogu koristiti više puta u aplikaciji
  • doprinosi boljoj struktuiranosti programa.

U C#-u aplikacija se struktuira pomoću klasa.
Sve metode moraju pripadati nekoj klasi ili zapisu.
Metoda je član klase koji izvršava određenu akciju - skup C# naredbi koje su objedinjene i imenovane.

Metode mogu biti:
  • metode instance
  • koje rade nad određenom instancom klase

- statičke metode
- koje obezbeđuju opštiju funkcionalnost, tj. ne zahtevaju da postoji instanca klase.

Kreiranje metoda

Neophodno je navesti sledeće:

ime

- metoda ne može imati isto ime kao i bilo koja promenljiva, konstanta ili neki drugi član koji je deklarisan unutar klase
- za imenovanje metoda važe ista pravila kao i za imenovanje promenljivih.

lista parametara

- navodi se unutar zagrada koje moraju biti navedene i ukoliko nema parametara

telo metode

- navodi se unutar vitičastih zagrada, čak i ukoliko se telo sastoji samo od jedne naredbe.

tip povratne vrednosti

- navodi se ispred imena metode
- ukoliko metoda ne vraća vrednost navodi se void.

```
TipPovratneVrednosti ImeMetode (ListaParametara)
{ TeloMetode }
```

Pozivanje metoda

Nakon definisanja metode, ona se može pozvati iz same klase ili iz neke druge klase.

Pozivanje metoda iz same klase

- koristi se ime metode iza koje sledi lista parametara unutar zagrada

Pozivanje metoda iz drugih klasa:

- prvo se mora navesti ime klase čija se metoda poziva
- kada ne bi bilo navedeno ime klase kompajler bi tražio datu metodu unutar klase iz koje se poziva, i ukoliko ne bi postojala prijavio bi grešku.
- ukoliko se za metodu ne navede modifikator pristupa, kao podrazumevani modifikator će se koristiti private.
  - Ovakva meteoda se može koristiti isključivo unutar klase u kojoj je definisana.

Deklarisanje parametara i poziv metoda

- Parametri omogućavaju prenos informacija u i iz metode.
- Svaki parametar ima svoj tip i ime.
- Parametri se deklarišu navođenjem njihovih pojedinačnih deklaracija (odvojenih zarezima) unutar zagrada iza imena metode.

```
void Primer(int x, string y)
{ ... }
```

Poziv metoda sa parametrima:

- pri pozivu metode moraju biti obezbeđene vrednosti za parametre.
- ovo se može učiniti:
  - prilikom samog poziva
  - pre poziva

```
Primer(2,"Zdravo");
...
int a = 2;
string b = "Zdravo";
Primer(a,b);
```

Lokalne promenljive

Svaka metoda ima svoj skup lokalnih promenljivih. Ove promenljive se mogu koristiti samo unutar metode u kojoj su deklarisani.

Promenljive deklarisane unutar jedne metode su potpuno nezavisne od promenljivih koje su deklarisane unutar drugih metoda, čak iako imaju ista imena.
Memorija u kojoj se čuvaju lokalne promenljive se alocira svaki put kada se pozove metoda, i oslobađa nakon izvršavanja metode. To znači da se bilo koje vrednosti, koje su čuvaju u ovim promenljivima, neće zadržati od jednog poziva metode do drugog.
Lokalnoj promenljivoj se može dodeliti početna vrednost prilikom deklaracije, sve dok se ne dodeli početna vrednost, promenljiva je neinicijalizovana

```
void Primer(){
    int x;
    int y = 0;   // inicijalizacija
    ...
    Console.WriteLine(x); //Greska: Use of unassigned local variable 'x'
}
```

Zajedničke promenljive

Promenljive deklarisane unutar jedne metode su potpuno nezavisne od promenljivih koje su deklarisane unutar drugih metoda, čak iako imaju ista imena.
Memorija u kojoj se čuvaju lokalne promenljive se alocira svaki put kada se pozove metoda, i oslobađa nakon izvršavanja metode. To znači da se bilo koje vrednosti, koje su čuvaju u ovim promenljivima, neće zadržati od jednog poziva metode do drugog.

```
class Primer
{
    void Init()
    {
        int brojac = 0;
    }
    void Prebroj()
    {
        int brojac = 0;
        ++brojac;
    }
...
Init();
Prebroj();
```

Ovaj problem se može rešiti tako što će se promenljiva brojač deklarisati na nivou klase, a ne metode.
Promenljivu brojač dele sve metode date klase.
Ovaj problem se može rešiti tako što će se promenljiva brojač deklarisati na nivou klase, a ne metode.
Promenljivu brojač dele sve metode date klase.

```
class Primer
{
    int brojac;
    void Init()
    {
        brojac = 0;
    }
    void Prebroj()
    {
        ++brojac;
    }
...
Init();
Prebroj();
```

Naredba return

Može se koristiti da bi se izvršavanje momentalno vratilo iz metode na mesto poziva. Ukoliko se izostavi, izvršavanje se vraća na mesto poziva nakon izvršenja poslednje naredbe u metodi.

```
void Primer(){
    int broj = 5;
    if (broj < 10)
        return;
    Console.WriteLine("Dvocifren broj");
}
```

Može se navesti i više return naredbi unutar jedne metode.
Ukoliko je u definiciji metode naveden tip podatka koji se vraća mora se dodati bar jedna return naredba kojom se vraća vrednost iz metode.

```
void Primer(){
    if (broj % 2 == 0){
        Console.WriteLine("deljiv sa dva");
        return; }
    if (broj % 3 == 0){
        Console.WriteLine("deljiv sa tri");
        return; }
}
```

Povratne vrednosti

• Da bi se vratila vrednost neophodno je:
• prilikom deklaracije metode definisati tip povratne vrednosti
    • umesto ključne reči void navodi se tip povratne vrednosti.
• dodati return naredbu u metodu uz vrednost (ili izraz) koji treba vratiti
    • ovim će se postaviti povratna vrednost, momentalno prekinuti izvršavanje tekuće metode i vratiti izraz kao povratna vrednost metode.
• prilikom poziva metode prihvatiti vrednost

```
class Primer
{
    public int Brojac()
    {
        int a = 0;
        return ++a;
    }
    public void Proba()
    {
        int x = Brojac();
    }
}
```

Prenos parametara

• Return naredbom metoda može vratiti isključivo jednu vrednost.
• Ukoliko je neophodno vratiti više od jedne vrednosti:
    • može se vratiti referenca na niz, klasu ili zapis (koji mogu da sadrže više vrednosti)
    • mogu se koristiti mehanizmi za prenos parametara.
• Mehanizmi za prenos parametara:

Prenos preko vrednosti	ulazni parametri	Podaci se mogu preneti u metodu, ali se ne mogu preneti iz nje
Prenos preko referenci	ulazni i izlazni parametri	Podaci se mogu preneti u metodu, i iz metode
Prenos preko izlaza	izlazni parametri	Podaci se mogu preneti iz metode, ali se ne mogu preneti u nju

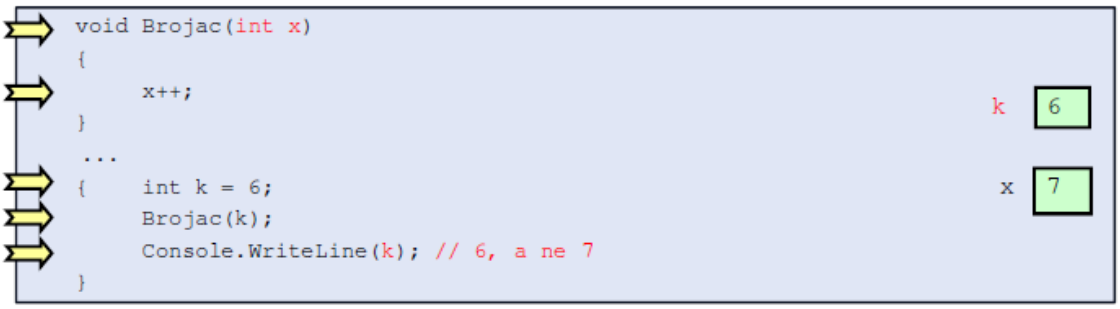
Mehanizmi za prenos parametara

U pricipu, argumenti se u metodu mogu preneti:
preko referenci
    • metoda referencira originalne promenljive
    • na originalne promenljive utiče bilo kakva izmena unutar pozvane metode
preko vrednosti
    • metoda referencira kopije originalnih promenljivih
    • originalne promenljive se ne menjaju bez obira na to kakve izmene nad njom vrši pozvana metoda (menjaju se samo kopije).

U C#-u se svi parametri prenose po vrednosti, ukoliko se ne navede drugačije.
NAPOMENA: kod stringova, iako su referentnog tipa, bilo kakva izmena nad stringom u metodi neće uticati na originalni string.
Sve promenljive moraju biti inicijalizovane pre nego što se mogu preneti u metodu, bez obzira na to da li se prenose preko vrednosti ili reference.

Prenos preko vrednosti

• Vrednosni parametar se navodi tako što se navede ime tipa, iza koga sledi ime promenljive.
• Kada se pozove metoda zauzima se nova memorijska lokacija za svaki vrednosni parametar. Vrednosti odgovarajućih izraza se kopiraju u njih.
• Unutar metode se može promeniti vrednost parametra, a da to ne utiče ni na jednu promenljivu izvan metode.
• Prosleđenja vrednost mora biti istog ili kompatibilnog tipa sa tipom u deklaraciji parametra. Izrazi koji su navedeni za svaki vrednosni parametar moraju biti istog tipa kao i tip koji je naveden u njegovoj deklaraciji, ili tipa koji se implicitno može konvertovati u taj tip.



Prenos preko reference

• Referentni parametar predstavlja referencu na memorijsku lokaciju.
• Za razliku od vrednosnih parametara, oni ne zauzimaju novu memorijsku lokaciju, već pokazuju na istu onu lokaciju na koju pokazuje promenljiva prosleđena pozivom metode.
• Referentni parametri se deklarišu navođenjem ključne reči ref ispred tipa.
• Ključna reč ref se odnosi samo na parametar ispred koga je navedena, a ne na celu listu parametara.
• Prilikom poziva metode takođe se mora navesti ključna reč ref

```
void Primer(ref int x, long y){ ... }
void Primer(ref int x, ref long y){  ... }
...

int p =10;
long q = 15;

Primer(p, q)              // Greška
Primer(ref p, q);         // OK
Primer(ref p, ref q);     // OK
```

• Vrednost koja se prosleđuje u pozivu metode mora biti istog tipa, kao i tip naveden u definiciji metode, i mora biti promenljiva (ne može biti konstanta ili vrednost izraza).

```
void Primer(ref long x)
{
    x++;
}
   ...
 int y = 10;
 Primer(ref y);  // Greška
```

Referentnom parametru se mora dodeliti vrednost pre poziva metode.

```
long y;
Primer(ref y); // Greška
long y = 10;
Primer(ref y); // OK
```

Ukoliko se promeni vrednost referentnog parametra promeniće se i promenljiva koja je prosleđena pozivom metode. Obe predstavljaju referencu na istu lokaciju u memoriji.

Izlazni parametri

• Koriste se kada je potrebno vrati više od jedne vrednosti iz metode. Mogu samo da prosleđuju podatke iz metode (ne mogu ih proslediti u metodu).
• Kao i referentni parametri, predstavljaju referencu na memorijsku lokaciju, koja je prosleđena u pozivu metode
• Izlazni parametri se deklarišu navođenjem ključne reči out ispred tipa
• Ključna reč out se odnosi samo na parametar ispred koga je navedena, a ne na celu listu parametara.
• Prilikom poziva metode mora se navesti ključna reč out ispred promenljive koja se prosleđuje.

```
void Primer(out int x, long y){ ... }
void Primer(out int x, out long y){  ... }
...
int p = 15;
long q = 20;
Primer(p, q)              // Greška
Primer(out p, q);         // OK
Primer(out p, out q);     // OK
```

C# zahteva da sve promenljive budu inicijalizovane nekom početnom vrednošću pre nego što se mogu referencirati. U prethodnom početne vrednosti promenljivih koje se prenose su nebitne (metoda ih ne može koristiti)
Način da se zaobiđe ovaj zahtev C# kompajlera je da se ispred ulaznih parametara metode navede ključna reč out. Tretira se kao neinicijalizovana lokalna promenljiva. Promenljivoj koja je obezbeđena za izlazni parametar ne mora biti dodeljena vrednost pre poziva metode

```
int y;
Primer(out y); // OK
```

Izlaznom parametru je neophodno dodeliti vrednost unutar metode. U suprotnom metoda se neće iskompajlirati.

```
void Primer(out int x, out long y)
{
    x = x + 1;    // Greske: 1. Use of unassigned local variable 'x'
}                 //         2. The out parameter 'y' must be assigned to before
                  //            control leaves the current method
```

Pošto se promenljiva prenosi preko reference, sve promene koje metoda izvrši nad tom promenljivom ostaju i nakon povratka kontrole na mesto poziva. Obe predstavljaju referencu na istu lokaciju u memoriji.

Preporuke za prenos parametara

Prilikom izbora načina prenosa parametara treba voditi računa o mehanizmu prenosa i njegovoj efikasnosti.

Mehanizmi prenosa

• prenos preko vrednosti je najčešći

• vrednosne parametre ne treba koristiti ukoliko je potrebno proslediti informacije iz metode

• ukoliko je potrebno proslediti podatke izvan metoda mogu se koristiti return naredba, referentni parametri ili izlazni parametri

• ukoliko je potrebno vratiti samo jednu vrednost koristiti naredbu return sa povratnom vrednošću

• ukoliko je potrebno vratiti više vrednosti koristiti ref i/ili out parametre

• ref treba koristiti samo kada podatke treba proslediti u oba pravca

Efikasnost

• prenos preko vrednosti je generalno najefikasniji. Jednostavni tipovi (kao što su int i long) se najefikasnije prenose preko vrednosti.

• Za kompleksne tipove podataka, prenos preko referenci je efikasniji zbog velike količine podataka koju bi trebalo kopirati ukoliko bi se prenosile preko vrednosti.

Referentni tipovi kao parametri

Promenljive referentnog tipa se kao parametri metoda mogu preneti na tri načina:

- preko vrednosti
- preko reference (ref)
- kao izlazni parametar (out)

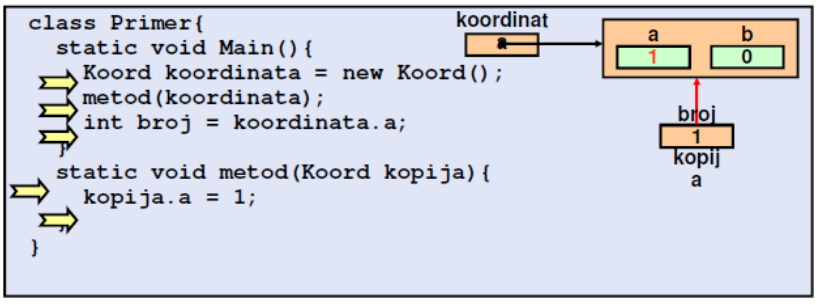
Preko vrednosti

• Metoda pravi kopiju reference

• dve reference (original i kopija) referenciraju isti objekat

• originalna referenca se ne može promeniti

• promene nad referenciranim objektom će biti zapamćene



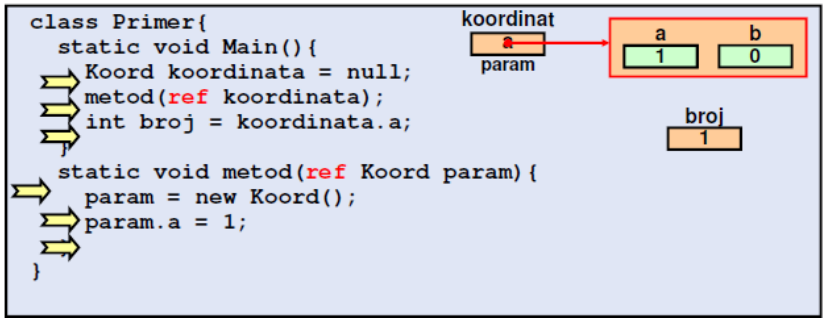
Preko reference

• Metoda ne pravi kopiju reference

• radi se sa originalnom referencom

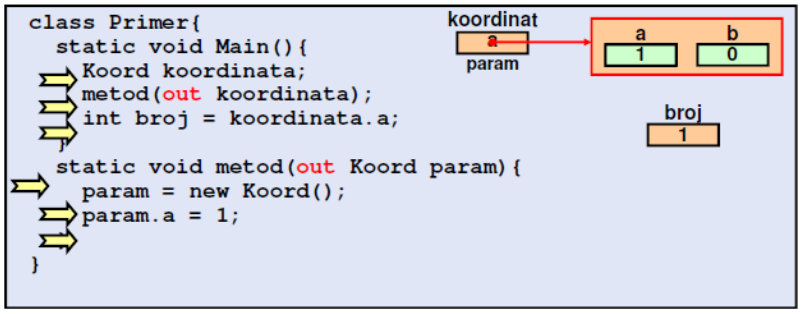
• originalna referenca se može promeniti

• promene nad referenciranim objektom će biti zapamćene



Kao izlazni parametar

- Metoda ne pravi kopiju reference
- radi se sa originalnom referencom
- originalnoj referenci se mora dodeliti vrednost



Zapis kao parametar metoda

Ukoliko su zapisi veliki njihovim kopiranjem se smanjuju performance. Performanse su bolje ukoliko se zapisi prosleđuju kao ref parametric

Promenljivi broj parametara

- Ponekad je korisno da metoda može da primi promenljiv broj parametara.
- Koristi se ključna reč params da bi se navela lista parametara promenljive dužine.

OGRANIČENJA:

- može se navesti samo jedan params parametar po metodi
- on se mora navesti na kraju liste parametara
- deklariše se kao jednodimenizionalni niz.

```
int Primer(params int[] brojevi)
{
    int ukupno, i;
    for(i = 0, ukupno = 0; i < brojevi.Length; i++)
         ukupno += brojevi[i];
    return ukupno;
}
```

Na osnovu svojstva niza Length može se odrediti broj prosleđenih parametara. Sve vrednosti koje se navode moraju biti istog tipa pošto je tip parametra params uvek niz.

Pri pozivu metode vrednosti se mogu proslediti params parametru na dva načina (u oba slučaja parametar se tretira kao niz):

- preko liste elemenata, odvojenih zarezima (lista može biti prazna)
- preko niza

```
int x;
x = Primer(63, 21, 84);
x = Primer(new x[] {63, 21, 84});
```

Kod params parametra, pravi se kopija podataka, pa promene vrednosti unutar metode neće uticati na vrednosti izvan metode. Uvek ih treba prenositi preko vrednosti.

Rekurzivne metode

- Metoda može samu sebe da poziva
  - direktno
  - indirektno
- Ova mogućnost se naziva rekurzija.
- Korisne su za manipulisanje složenim strukturama podataka kao što su liste ili stabla.
- Metode u C#-u mogu da budu međusobno rekurzivne. Dozvoljena je situacija u kojoj metoda A može da zove metodu B, a i metoda B može da zove metodu A.
- Rekurzivna metoda mora imati uslov izlaza koji obezbeđuje da se iz metode može vratiti bez daljih poziva.

```
ulong Fibonacci(ulong n)
{
    if (n <= 2)    // uslov izlaza
         return 1;
    else
         return Fibonacci(n-1) + Fibonacci(n-2);
}
```

Potpis metode

Formiraju potpis metode	Ne utiču na potpis metode
Ime metode	Imena parametara
Broj parametara koji metoda prima	Tip povratne vrednosti metode
Tip parametara	
Modifikator parametara (ref ili out)	

int Primer(int n) { ... }

```
int Primer(int x) { ... } // isti potpis
string Primer(int n) { ... } // isti potpis
int Primer(ref int n) { ... }
int Primer(int n, int p) { ... }
```

Preklapanje metoda

Da bi se preklopile metode, one se jednostavno deklarišu sa istim imenom, ali različitim brojem ili tipom parametara.
NAPOMENA: nije dozvoljno da se dve metode razlikuju samo po tome da li je određeni parametar deklarisan kao ref ili out !
C# pravi razliku između ovih metoda upoređivanjem njihovih lista parametara.

```
class Primer {
    int Zbir(int a, int b)
    {
         return a+b;
    }
    int Zbir(int a, int b, int c)
    {
         return a+b+c;
    }
    ...
    {
         Console.WriteLine(Zbir(1, 2) + Zbir(1, 2, 3));
    }
}
```

Upotreba preklapanja metoda

U opštem slučaju preklapanje metoda treba razmatrati:

• kada je potreban određeni broj metoda koje primaju različite parametre (broj ili tip), ali u suštini rade istu stvar.

• npr. Console.WriteLine()

• kada je potrebno dodati novu funkcionalnost postojećem kôdu.

• pošto C# ne dozvoljava eksplicitnu dekaraciju default parametara u metodama, a ni opcionih parametara, neophodno ih je simulirati preklapanjem metoda.

```
class Primer {
int Uvecaj(int a){
return Uvecaj(a, 1);
}
int Uvecaj(int a, int b) {
return a+b;
}
}
```

PREPORUKA: Preklapati samo metode koje su srodne po funkcionalnosti, a razlikuju se samo po broju ili tipu potrebnih parametara.

```
class Primer
{
   enum DobaDana {Jutro, Vece}
   void Pozdrav(){
        Console.WriteLine("Zdravo"); }
   void Pozdrav(string ime) {                // slična metoda
        Console.WriteLine("Zdravo" + ime); }
   void Pozdrav(string ime, DobaDana dd) {   // nova funkcionalnost
        string poruka = "";
        switch(dd)
        { case DobaDana.Jutro: poruka = "Dobro jutro";
                               break;
          case DobaDana.Vece:  poruka = "Dobro Vece";
                               break;
        }
        Console.WriteLine(poruka + "" + ime);
     }
     ...
   Pozdrav();
   Pozdrav("Pero");
   Pozdrav("Pero", DobaDana.Jutro);
}
```

Main() metoda

• Izvršavanje C # programa počinje kod metode koja se zove Main().

• Ona mora biti statička metoda klase (ili zapisa), i mora vraća vrednost tipa int ili void.

• Iako je uobičajeno da se public modifikator eksplicitno navede, za ovu metodu nije neophodno njegovo navođenje.

• po definiciji ova metoda mora biti pozvana izvan programa, pa zapravo nije bitno koji joj je nivo pristupa dodeljen
• ona će se izvršavati čak iako je označena kao privatna.
• Prilikom kompajliranja C# konzole ili Windows aplikacije, kompajler po default-u traži tačno jednu Main() metodu, unutar bilo koje klase, čiji potpis odgovara navedenom, i postavlja tu metodu za tačku ulaza u program.
• Ukoliko postoji više Main() metoda, kompajler će javiti grešku.
• međutim, može se kompajleru eksplicitno reći koju od ovih metoda da koristi kao tačku ulaza u program, korisćenjem /main switch-a (prekidača) zajedno sa punim kvalifikovanim imenom (uključujući i namespace) klase unutar koje se nalazi ta metoda.
• npr. csc MainPrimer.cs /main:FON.PJ.Primer

Naredbe

Blok naredbi
• Grupa naredbi obuhvaćena vitičastim zagradama { } naziva se **blok**.
  • može da sadrži jednu ili više naredbi, odnosno jedan ili više blokova.
  • blok koji se nalazi unutar drugog bloka naziva se ugnježđeni blok.
• Promenljiva deklarisana unutar bloka naziva se **lokalna promenjiva**
  • mogu se definisati na bilo kom mestu unutar bloka.
  • opseg važnosti je od trenutka njene deklaracije do kraja blok.

{ // pocetak bloka
```
int i; // deklaracija lokalne promenljive
} // kraj bloka
```

U spoljašnjem i unutrašnjem bloku ne mogu biti deklarisane promenjive sa istim imenom.

```
{           // početak spoljašnjeg bloka
  int i;    // deklaracija lokalne promenljive
  {         // početak unutrašnjeg bloka
    int i;  // deklaracija lokalne promenljive
  }         // kraj unutrašnjeg bloka
}           // kraj spoljašnjeg bloka
```

Dozvoljena je deklaracija dve promenjive sa istim imenom u blokovima na istom nivou.

```
{           // početak bloka
  int i;    // deklaracija lokalne promenljive
}           // kraj bloka
{           // početak bloka
  int i;    // deklaracija lokalne promenljive
}           // kraj bloka
```

Upravljačke strukture
• Selektivno se izvršavaju naredbe na osnovu vrednosti uslova.
UPRAVLJACKE STRUKTURE SELEKCIJE - naredbe if i switch
• Iterativno se izvršavaju naredbe dok je vrednost uslova true.
ITERATIVNE UPRAVLJACKE STRUKTURE - naredbe while, do i foreach
• Koriste se za bezuslovan prelazak na drugu naredbu.

if naredba
```
if (uslov)
naredba(blok naredbi)
else // opcioni
```

```
naredba(blok naredbi)
```

Algoritam izvršavanja
1. Ispituje se vrednosti uslova.
2. Ukoliko je true kontrola se prenosi na prvu naredbu then bloka. Kada se stigne do kraja tog bloka, kontrola se prenosi na kraj if naredbe.
3. Ukoliko je false, a postoji else blok, kontrola se prenosi prvu naredbu else bloka. Kada se stigne do kraja tog bloka, kontrola se prenosi na kraj if naredbe.
4. Ukoliko je false, a ne postoji else blok, kontrola se prenosi na kraj if naredbe

Navođenje vitičastih zagrada nije neophodno u slučaju da blok naredbi sadrži samo jednu naredbu.

Kaskadne if naredbe

```
if ( uslov1 )
naredba1;
else if ( uslov2 )
naredba2;
else
naredba4;
```

Ugnježdene if naredbe

```
if ( uslov1 )
if ( uslov2 )
naredba1;
else
naredba2;
```

else se odnosi na najbliži if.

switch naredba

Switch naredba je naredba višestrukog grananja
- izbor jedne grane izvršavanja iz skupa međusobno isključivih grana.
- alternativa ugnježdenim if naredbama

Sintaksa

```
switch ( selektor )
{
   case konstanta1:
        naredba1(blok1);
        break;
   case konstanta2:
        naredba2(blok2);
        break;
   default:
        naredba3(blok3);
        break;
}
```

Tip selektora može biti: char, enum, string ili neki celobrojni tip. Može se navesti i neki drugi tip samo ukoliko postoji tačno jedna korisnički definisana implicitna konverzija.
Ukoliko je u pitanju string kao konstanta se može navesti i NULL vrednost.

- case i default labele predstavljaju tačke u koje se kontrola toka programa može preneti u zavisnosti od vrednosti selektora
- Kao poslednja naredba svakog case bloka navodi se break ili neka druga naredba bezuslovnog prelaska.
- Nije neophodno navođenje vitičastih zagrada kako bi se više naredbi obuhvatilo u case blok.
- Nije dozvoljeneno navođenje više od jedne konstante u jednoj labeli.
- Redosled navođenja case blokova nije bitan. Default blok se može navesti kao prvi.

```
switch ( izbor )
{
case default: ...;
```

```
case 1: ...;
case 2: ...;
}
```

Ista vrednost konstante se ne može navesti više puta.

```
switch ( izbor )
{
case 1: ...;
case 1: ...; // greška
default: ...;
}
```

Ne mogu se navesti dve default labele.

```
switch ( izbor )
{
case 1: ...;
default: ...;
default: ...; // greška
}
```

Algoritam

1. Ukoliko je vrednosti selektora jednaka konstanti neke od case labela kontrola se prenosi na prvu naredbu bloka koji odgovara toj labeli.
2. Ukoliko vrednost selektora nije jednaka nijednoj od konstanti u case labelama, a postoji default labela, kontrola se prenosi na prvu naredbu default bloka.
3. Ukoliko vrednost selektora nije jednaka nijednoj od konstanti u case labelama, a ne postoji default labela, kontrola se prenosi na kraj switch naredbe

Ukoliko se izvrši neki od case blokova nijedan sledeći case blok se ne može izvršiti osim u slučaju korišćenja goto naredbe kojom se kontrola izvršenja prenosi na neki od tih blokova. Jedini izuzetak je u slučaju grupisanja kada se ne navode naredbe za taj case blok (on je prazan).

```
swit ch (selektor)
{
case 1: ...; goto case 3;
case 2: ...; break;
case 3: ...; break;
}
```

izuzetak je u slučaju grupisanja.

```
switch (selektor)
{
case 1:
case 2: ...; break;
case 3: ...; break;
}
```

while naredba

Sintaksa

```
while (uslov)
naredba(blok naredbi);
```

- Petlja sa preduslovom
- Uslov mora biti logički izraz.
- Ponavlja se izvršavanje bloka naredbi sve dok uslov ima vrednost true.
- Ovu naredbu treba koristiti kada unapred nije poznat broj iteracija.

Algoritam

1. Ispituje se vrednosti uslova.
2. Ukoliko je true, kontrola se prenosi na prvu naredbu while bloka. Kada stigne do kraja tog bloka, ona se implicitno prenosi na početak while naredbe i vrši se ponovno ispitivanje uslova.
3. Ukoliko je false, kontrola se prenosi na kraj while naredbe.

Pošto se uslov ispituje na početku petlje, može se desiti da se naredbe unutar nje nikad ne izvrše.

```
int i = 1;
while (i < 10) // (i > 10) petlja se nece izvršiti
{
  Console.WriteLine(i);
  i++;
}
```

do naredba
Sintaksa

```
do
naredba(blok naredbi);
while (uslov);
```

- petlja sa postuslovom
- uslov mora biti logički izraz.
- ponavlja se izvršavanje bloka naredbi sve dok je uslov ima vrednost true.
- ovu naredbu treba koristiti kada je potrebno da se blok naredbi barem jedanput izvrši.

do naredba
Algoritam
1. Kontrola se prenosi na do naredbu.
2. Kada tok kontrole stigne do kraja tog bloka ispituje se vrednost uslova
3. Ukoliko je true, kontrola se ponovo prenosi na početak do naredbe.
4. Ukoliko je false, kontrola se prenosi na prvu naredbu izvan petlje.

Pošto se uslov ispituje na kraju, naredbe unutar petlje će se bar jedanput izvršiti.

```
int i = 1;
do
{
  Console.WriteLine(i);
  i++;
}
while (i < 10); // (i > 10) petlja se izvršava jedanput
```

for naredba
Sintaksa

```
for (inicijalizator, uslov, iterator)
naredba(blok naredbi);
```

- petlja sa preduslovom.
- koristiti se kada je unapred poznat broj iteracija.
- inicijalizator
  - izraz koji se izračunava pre početka rada petlje
  - obično se inicijalizuje lokalna promenjiva poznata kao brojačka promenjiva.
- uslov
  - logički izraz koji se proverava pre svake iteracije.
  - da bi se izvršila iteracija njegova vrednost mora biti true.
- iterator
  - izraz koji se izračunava posle svake iteracije

- običnо se uvećava brojačka promenjiva.

• inicijalizator, uslov i iterator su opcioni.

Algoritam
1. Inicijalizuju se brojačke promenjive i ovaj korak se vrši samo jednom.
2. Ispituje se vrednosti uslova.
3. Ukoliko je true, kontrola se prenosi na prvu naredbu bloka. Kada stigne do kraja tog bloka kontrola se implicitno prenosi na početak for naredbe kada se ažuriraju brojačke promenjive i vrši ponovno ispitivanje uslova.
4. Ukoliko je false, kontrola se prenosi

Pošto se uslov ispituje na početku petlje može se desiti da se naredbe unutar nje nikad ne izvrše.
```
for (int i = 1; i < 10; i++) //(i > 10) petlja se nece izvršiti
Console.WriteLine(i);
```

Ukoliko se ne navede uslov on se implicitno smatra tačnim što može izazvati beskonačnu petlju.
```
for (int i = 0 ; ; i++)
{
... ;
}
```

Jedna for naredba može da sadrži više kontrolnih promenjivih
```
for (int i = 0, j = 10 ; i < j; i++, j--)
{
... ;
}
```

foreach naredba
Omogućava iteriranje kroz sve elemente kolekcije

Sintaksa
```
foreach (tip iteraciona_promenjiva in kolekcija)
naredba(blok naredbi);
```

• iteraciona promenjiva predstavlja element kolekcije kroz koju se iterira.
• njena vrednost se ne može eksplicitno promeniti.

```
fore ach (int element in kolekcija) element++; // greška
```

• mora postojati eksplicitna konverzija iz tipa elementa kolekcije u tip iteracione promenjive.
• System.Array tip je kolekcija pa se može napisati sledeća foreach naredba

```
int [,] niz = { {1,2},{3,4},{5,6} };
foreach (int element in niz)
Console.WriteLine(element);
```

• uslov je logički izraz koji se proverava pre svake iteracije.
• da bi se izvršila iteracija njegova vrednost mora biti true.
• iterator je izraz koji se izračunava posle svake iteracije
• običnо se uvećava brojačka promenjiva.

goto naredba
Prenosi tok kontrole na naredbu koja je označena labelom.

```
goto labela;
...
labela: ...;
```

• Ukoliko ne postoji navedena labele ili naredba goto nije u njenom opsegu prijaviće se greška.
• Naredbom goto tok kontrole se može preneti izvan ugnježdenog bloka, ali ne i unutar njega.

```
for (...)
{
...;
if (uslov) goto kraj;
}
kraj:
...;
```

• Naredbom goto ne može se izaći iz finally bloka.
• labela na koju se prenosi kontrola mora biti unutar finally bloka

```
try
{
...;
}
catch
{
...;
}
finaly
{
...;
goto labala;
...;
labela:
...;
}
```

• Ukoliko se naredba goto nađe unutar try bloka koji ima finally blok.
• pre prenosa kontrole će se izvršiti finally blok.

```
try
{
...;
goto labela;
...;
}
finaly
{
...;
}
...;
labela:
...;
```

break naredba

• obezbeđuje iskakanje iz switch, while, do, for i foreach naredbi.
• kontrola se prenosi na prvu sledeću naredbu.
• tok kontrole se ne sme preneti izvan finally bloka.
• ukoliko se nalazi u ugnježdenoj petlji, njenim izvršavanjem se tok kontrole prenosi na kraj unutrašnje petlje.

```
while (uslov)
{
  while (uslov)
  {
    ...;
    if (uslov) break;
    ...;
  }
  ...;
}
```

ukoliko se nađe unutar try bloka, pre prenosa kontrole će se izvršiti pridruženi finally blok.

```
while (uslov)
{
 try
 {
  ...;
  break;
  ...;
 }
 finaly
 {
  ...;
 }
}
...;
```

continue naredba

• započinje izvršavanje nove iteracije tekuće while, do, for i foreach naredbe.
• iskače se iz samo iz tekuće iteracije, što znači da se izvršavanje nastavlja na početku sledeće iteracije, a ne izvan cele petlje.
• tok kontrole se ne sme preneti izvan finally bloka.
• ukoliko se nalazi u ugnježdenoj petlji, njenim izvršavanjem se tok kontrole prenosi na početak unutrašnje petlje.

```
while (uslov)
{
  while (uslov)
  {
    ...;
    if (uslov) continue;
    ...;
  }
}
```

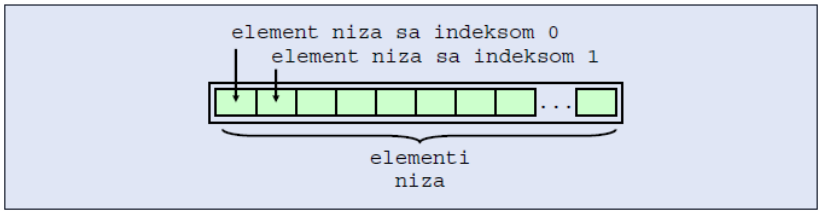
ukoliko se nađe unutar try bloka, pre prenosa kontrole će se izvršiti pridruženi finally blok.

```
while (uslov)
{
 try
 {
  ...;
  continue;
  ...;
 }
  finaly
 {
 ...;
 }
}
```

Nizovi

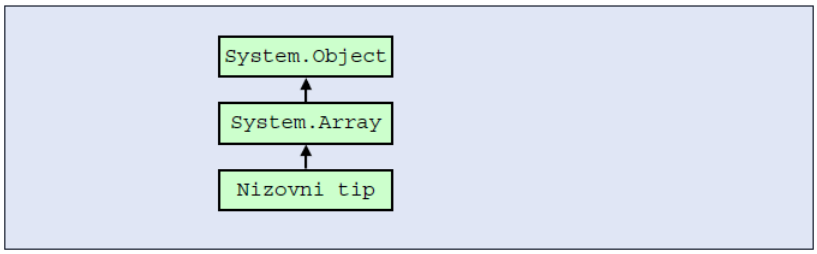
Šta je niz?

Niz je struktura podataka koja omogućava grupisanje konačnog broja promenljivih istog tipa.



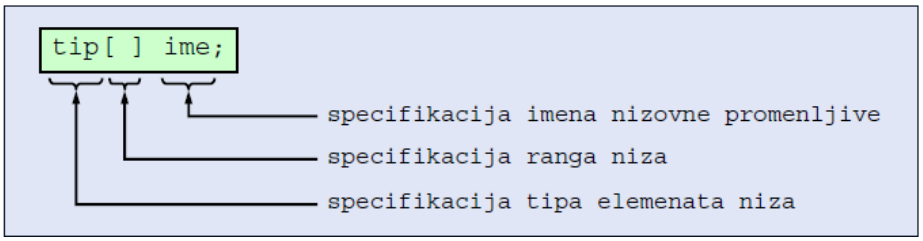
Promenljive sadržane u nizu se nazivaju elementi niza:
• istog su tipa (tip elemenata niza)
• smeštene su u susednim memorijskim lokacijama
• pristup im je omogućen preko indeksa
• brzina pristupa bilo kom elementu niza je ista

Nizovni tip
Nizovi su uvek referentni tipovi ! Nizovni tipovi se implicitno izvode iz System.Array koji je izveden iz System.Object tipa.



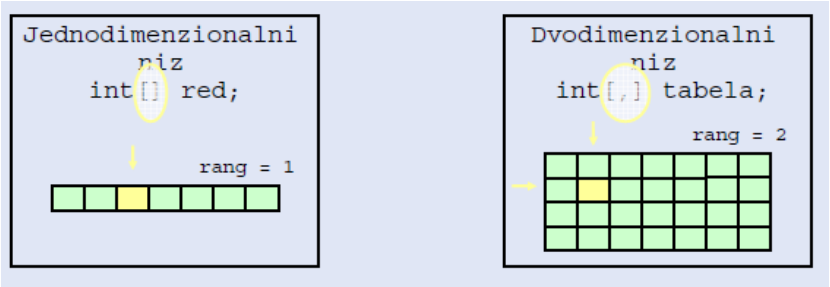
Promenljive nizovnog tipa sadrže referencu na niz, a ne niz.

Deklaracija niza
Za deklaraciju niza koristi se slična sintaksa kao i za deklaraciju bilo koje nenizovne promenljive, s tom razlikom što se koriste uglaste zagrade [ ].
Da bi se deklarisala nizovna promenljiva potrebno je:
• navesti tip elemenata niza, zatim
• uglaste zagrade kojima se specificira rang niza, i na kraju
• ime nizovne promenljive iza koga sledi tačka zarez.



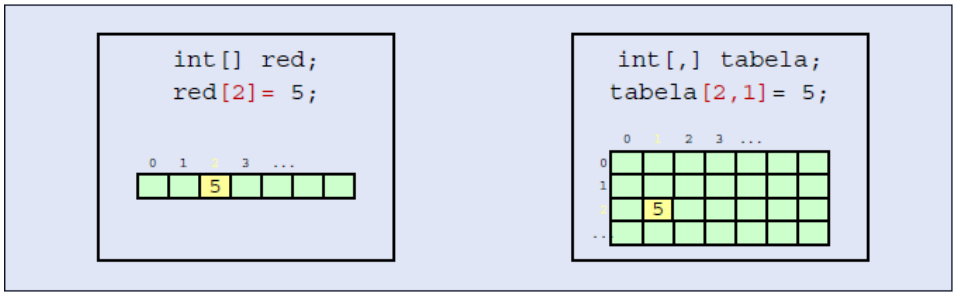
Tip elemenata niza može biti bilo koji tip uključujući i nizovni tip. Podržani su jednodimenzionalni, dvodimenzionalni i nizovi nizova

Rang niza
Rang niza ↔broj dimenzija niza ↔broj indeksa vezanih za element niza



Dodavanjem zareza povećava se rang niza za 1.
Dužina dimenzije se ne navodi prilikom deklaracije nizovne promenljive.

Pristup elementima niza
Da bi se pristupilo određenom elementu niza potrebno je navesti naziv nizovne promenljive, zatim uglaste zagrade [ ] unutar kojih se navodi odgovarajuća vrednost indeks.
Za jednodimenzionalne nizove navodi se jedan indeks, za dvodimenzinalne dva indeksa odvojeni zarezom, odnosno ako niz ima *n* dimenzija potrebno je navesti i *n* indeksa odvojenih zarezima.



Indeksi bez obzira na rang niza počinju od 0. Postoji tehnika kojom se donja granica niza može promeniti, ali se ona koristi veoma retko. (Array.CreateInstance)

Brži pristup eleme

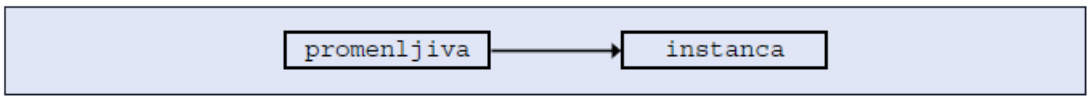
- Prilikom pristupa elementima niza vrši se provera vrednosti indeksa.
- ukoliko je izvan dozvoljenog opsega => System.IndexOutOfRangeException.
- Provera vrednosti indeksa utiče na performanse.
- Primer u kome se ne vrši kontrola indeksa
- veća brzina pristupa – veća mogućnost da se napravi greška !

```
class Primer
{
  unsafe static void Main()
  {
    int[] niz = new int[] {1,2,3,4};
    fixed (int* pElement = &niz[0]);
    {
      for (int x = 0; d = niz.Length; x <= d; x++)
         Console.WriteLine(element[x]);
    }
  }
}
```

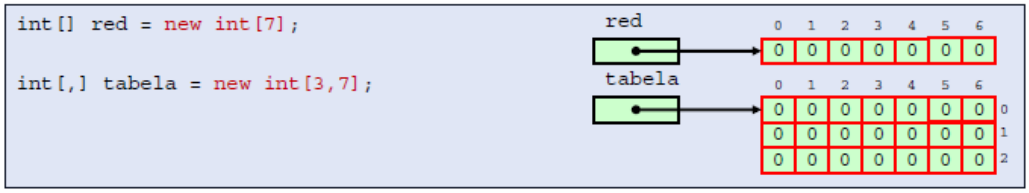
U prethodnom primeru je napravljena greška. Pristup nepostojećem elementu.

Kreiranje instance niza

Deklaracijom nizovne promenljive nije kreirana instanca niza, već samo promenljiva koja će sadržati referencu na instancu niza.



Koristi se ključna reč new.
Moraju se navesti dužine svih dimenzija niza.



Nakon kreiranja instance, kompajler implicitno inicijalizuje sve elemente niza na podrazumevanu vrednost. Koja će to vrednost biti zavisi od tipa elemenata niza, npr. za tip int podrazumevana vrednost je 0, za boolean je false, itd.

Inicijalizacija niza

Za inicijalizaciju instance niza koristi se inicijalizator niza.

Inicijalizator niza se sastoji od inicijalizatora promenljivih, odvojenih zarezima, koji su smešteni unutar vitičastih zagrada.

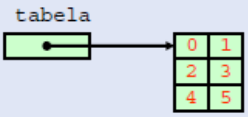
Inicijalizator promenljive je izraz, odnosno u slučaju višedimenzionalnih nizova ugnježđeni inicijalizator niza.

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

Inicijalizacijom niza moraju biti obuhvaćeni svi elementi niza.

Inicijalizacija višedimenzionalnog niza

```
int[,] tabela = new int[3,2];
tabela[0,0] = 0; tabela[0,1] = 1;
tabela[1,0] = 2; tabela[1,1] = 3;
tabela[2,0] = 4; tabela[2,1] = 5;

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



Svi elementi niza se eksplicitno moraju inicijalizovati bez obzira na rang niza

```
int[,] tabela= new int[3,2] { {0,1}, {2,3} }; // Greška
```

Određivanje dimenzija niza

Dimenzije niza se mogu odrediti

prilikom prevođenja programa (konstante) `int[] red= new int[7];`

prilikom prevođenja programa (promenljive)

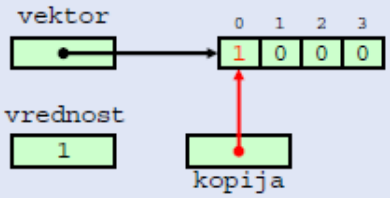
```
System.Console.WriteLine("Broj redova? : ");
string strRed= System.Console.ReadLine();
int red= int.Parse(strRed);

System.Console.WriteLine("Broj kolona? : ");
string strKolona= System.Console.ReadLine();
int kolona= int.Parse(strKolona);

int[,] tabela= new int[red,kolona];
```

Ograničenje. Ukoliko se koriste inicijalizatori, dimenzije niza se ne mogu zadati u toku izvršavanja programa.

Kopiranje nizovne promenljive

Kopiranjem nizovne promenljive ne kopira se instanca niza kopira se njena vrednost (referenca na instancu niza).

```
int[] vektor = new int[4];
int[] kopija;
kopija = vektor;
kopija[0] = 1;
int vrednost = vektor[0];
```

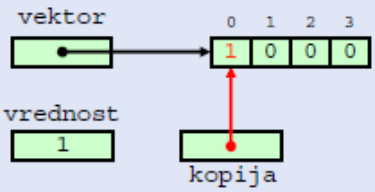


Nizovne promenljive vektor i kopija referenciraju istu instancu niza.

Niz kao parametar metoda

Nizovi se uvek prenose preko reference

```
class Primer
{
    static void Main()
    {
        int[] vektor = new int[4];
        metod(vektor);
        int vrednost = vektor[0];
    }
    static void metod(int[] kopija)
    {
        kopija[0]++;
    }
}
```

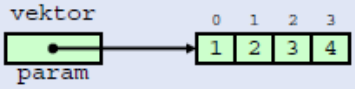


Promenljive vektor i kopija referenciraju istu instancu niza. Metoda radi sa originalom
Ukoliko metoda ne treba da radi sa originalom, kao parametar se prosleđuje kopija instance niza (Array.Copy metoda)

Parametar nizovnog tipa & ključna reč out

Parametar nizovnog tipa označen sa **out** mora biti određen unutar metode

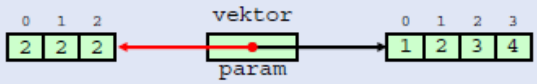
```
class Primer
{
    static void mojaMetoda(out int[] param)
    {
        param = new int[] {1,2,3,4};
    }
    static void Main()
    {
        int[] vektor;
        mojaMetoda(out vektor);
        System.Console.WriteLine("Elementi niza su :");
        foreach(int broj in vektor)
        {
            System.Console.WriteLine(broj);
        }
    }
}
```



Parametar nizovnog tipa & ključna reč ref

Parametar nizovnog tipa označen sa **ref** mora biti određen pre poziva metode

```
class Primer
{
    static void mojaMetoda(ref int[] param)
    {
        param = new int[] {2,2,2};
    }
    static void Main()
    {
        int[] vektor = new int {1,2,3,4};
        mojaMetoda(ref vektor);
        System.Console.WriteLine("Elementi niza su :");
        foreach(int broj in vektor)
        {
            System.Console.WriteLine(broj);
        }
    }
}
```

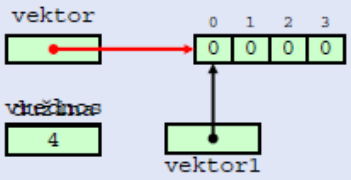


Niz kao povratna vrednost metoda

Prilikom specificiranja povratnog tipa metode ne smeju se navesti dimenzije niza, navodi se samo rang.

```
static int[] metoda(){...}
static int[,] metoda(){...}
static int[4] metoda(){...} // greška
```

```
class Primer
{
    static void Main()
    {
        int[] vektor = kreirajNiz(4);
        int vrednost = vektor[0];
    }
    static int[] kreirajNiz(int dužina)
    {
        int[] vektor1 = new int[dužina];
        return vektor1;
    }
}
```



Niz i naredba foreach

Naredba foreach omogućava pristup svim elementima niza bez korišćenja indeksa I provere njegove vrednosti i bez korišćenja izraza za pristup elementima.

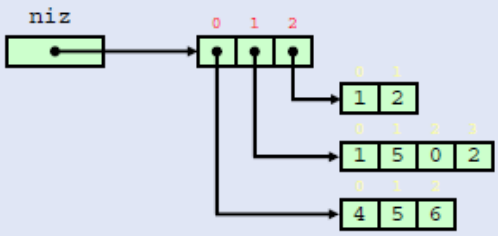
```
int[] vektor = new int[] {1,2,3,4};

for(int i = 0; i < vektor.Length; i++)
{
  System.Console.WriteLine(vektor[i]);
}

foreach(int broj in vektor)
{
  System.Console.WriteLine(broj);
}
```

```
int[,] tabela = new int[] { {1,2}, {3,4}, {5,6} };
foreach(int broj in tabela)
{
  System.Console.WriteLine(broj);
}
```

Zupčasti niz (Jagged array)

Zupčasti niz (niz nizova) je niz čiji su elementi nizovi.

```
int[][] niz;
niz = new int[3][];

niz[0] = new int[3] {4,5,6};
niz[1] = new int[4] {1,5,7,2};
niz[2] = new int[2] {1,2};

niz[1][2] = 0;
```



Performanse zupčastog niza čiji indeksi počinju od nule su iste kao i kod jednodimenzionalnog niza čiji indeksi počinju od nule. Zupčasti nizovi nisu u skladu sa specifikacijom opšteg jezika (CLS)

Svojstvo >> Rank

Vraća vrednost koja ukazuje na rang, odnosno ukupan broj dimenzija niza. Vrednost se ne može menjati

```
int[] niz1 = new int[5];
int[,] niz2 = new int[2,3];
int[,,] niz3 = new int[2,3,2];
System.Console.WriteLine(niz1.Rank); // 1
System.Console.WriteLine(niz2.Rank); // 2
System.Console.WriteLine(niz3.Rank); // 3
public int Rank {get;}
```

Metoda >> Clear()

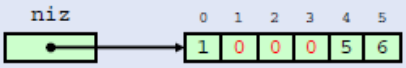
Metodom Clear navedenim elementima niza se dodeljuju podrazumevane vrednosti.

x – niz nad kojim se radi

i – indeks elementa od kog se počinje

d – broj obuhvaćenih elemenata

```
int[] niz = new int[6] {1,2,3,4,5,6};
System.Array.Clear(niz,1,3);
```



Metoda >> Copy()

Metodom Copy kopira se sekvenca elemenata jednog u drugi niz and performs type casting and boxing as required.

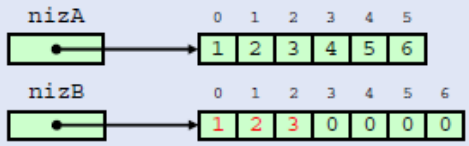
```
public static void Copy(Array, Array, int);
public static void Copy(Array, Array, long);
public static void Copy(Array, int, Array, int, int);
public static void Copy(Array, long, Array, long, long);
```

```
public static void Copy(Array x, Array y, int d);
```

```
public static void Copy(Array x, Array y, long d);
```

x – niz iz koga se kopira
y – niz u koji se kopira
d – broj obuhvaćenih elemenata

```
int[] nizA = new int[6] {1,2,3,4,5,6};
int[] nizB = new int[7];
System.Array.Copy(nizA, nizB, 3);
```

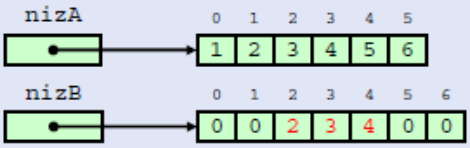



```
public static void Copy(Array x, int i, Array y, int j, int d);
```

```
public static void Copy(Array x, long i, Array y, long j, long d);
```

x – niz iz koga se kopira
i – indeks elementa niza x od kog se počinje
y – niz u koji se kopira
j – indeks elementa niza y od koga se počinje
d – broj obuhvaćenih elemenata

```
int[] nizA = new int[6] {1,2,3,4,5,6};
int[] nizB = new int[7];
System.Array.Copy(nizA, 1, nizB, 2, 3);
```



Metoda >> Sort()

Metodom Sort sortiraju se elementi jednodimenzionalnog niza.

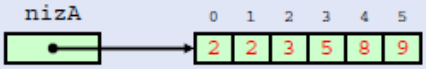
- elementi niza moraju da implementiraju interfejs IComparable
- ukoliko sortiranje nije uspešno završeno, rezultat je nedefinisan
- koristi se QuickSort algoritam
- ne vodi se računa o redosledu elemenata čija je vrednost ista

```
public static void Sort(Array);
public static void Sort(Array, int, int);
public static void Sort(Array, IComparer);
public static void Sort(Array, int, int, IComparer);
public static void Sort(Array, Array);
public static void Sort(Array, Array, int, int);
public static void Sort(Array, Array, IComparer);
public static void Sort(Array, Array, int, int, IComparer);
```

```
public static void Sort(Array x);
```

x – jednodimenzionalni niz koji se sortira

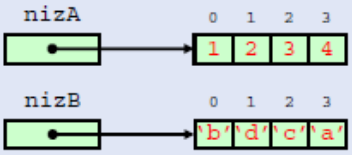
```
int[] nizA = new int[6] {8,2,5,3,2,9}
System.Array.Sort(nizA);
```




```
public static void Sort(Array x, Array y);
```

x – jednodimenzionalni niz po čijim vrednostima se sortira
y – jednodimenzionalni niz koji se sortira

```
int[] nizA = new int[4] {4,1,3,2};
char[] nizB = new char[4] {'a','b','c','d'};
System.Array.Sort(nizA, nizB);
```

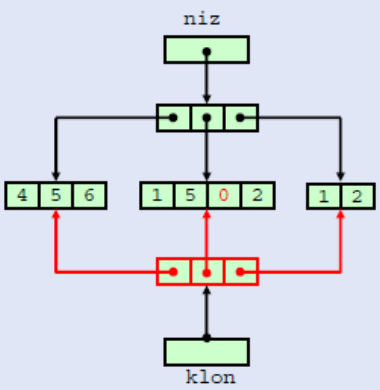


Metoda >> Clone()

Metodom Clone kreira se nova instanca niza čiji su elementi kopije elemenata kloniranog niza. Ukoliko su elementi niza referentnog tipa, objekti na koje oni imaju reference neće biti kopirani

```
public virtual object Clone();
```

```
int[][] niz  =
{
    new int[3] {4,5,6},
    new int[4] {1,5,7,2},
    new int[2] {1,2}
};
int[][] klon;
klon = (int[][])niz.Clone();
klon[1][2] = 0;
```



Metoda >> GetLength()
Vraća vrednost koja ukazuje na broj elemenata za određenu dimenziju niza. Prva dimenzija se označava sa 0.
`public int GetLength(int d);`
d – dimenzija niza

```
int[,] niz = { {0,1,2,3}, {4,5,6,7} };
System.Console.WriteLine(niz.GetLength(0)); // 2
System.Console.WriteLine(niz.GetLength(1)); // 4
```

Metoda >> IndexOf()
Vraća vrednost indeksa prvog elementa čija je vrednost jednaka traženoj. Ukoliko nepostoji element sa dtraženom vrednošću, vraća se -1. Može da se koristi samo nad jednodimenzionalnim nizovima.

```
public static int IndexOf(Array x, object a);
public static int IndexOf(Array x, object a, int p);
public static int IndexOf(Array x, object a, int p, int d);
```

x – niz koji se pretražuje
a – objekat koji se traži
p – indeks elementa od koga počinje pretraživanje
d – broj obuhvaćenih elemenata

```
int[] niz = { 0,5,2,3,5,4,6,7 };
Console.WriteLine(Array.IndexOf(niz, 5)); // 1
Console.WriteLine(Array.IndexOf(niz, 5, 2)); // 4
Console.WriteLine(Array.IndexOf(niz, 5, 5, 2)); //-1
```

Preporuke
Koristiti jednodimenzionalne nizove čije vrednosti indeksa počinju od 0. Postoje specifične instrukcije IL-a / optimizacija

CLS saglasnost
Msdn – array types in .NET strana 1

Izuzeci

Šta je obrada izuzetaka
• Greške su uvek moguće, mogu nastati u bilo kom trenutku izvršavanja programa.
• Dobar program je onaj koji je u stanju da identifikuje i obradi nastalu grešku.

Obrada izuzetaka je mehanizam koji:
• 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
• se bazira na predstavljanju izuzetaka pomoću objekata

Tradicionalna proceduralna

```
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.

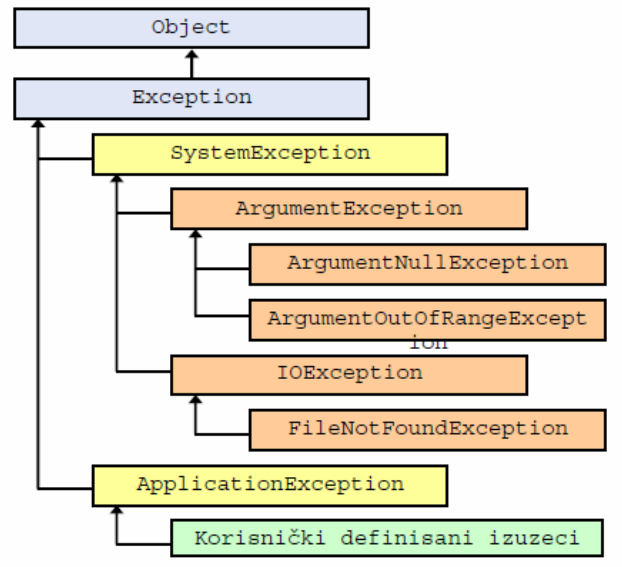
Nedostaci tradicionalne obrade

• 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 (primer: rezultat izvršavanja funkcije printf se često ne proverava. Ukoliko je rezultat -1, nastala je greška)
• Potreban je fleksibilniji mehanizam koji pruža dovoljno informacija o grešci

Klase Izuzetaka

• Da bi se prevazišao problem nedostatka informacija o nastalim greškama, .NET definiše niz različitih klasa izuzetaka.
• Izuzetak je objekat koji se kreira ili podiže (throw) kada dođe do određene greške i sadrži informacije koje bi trebale omoguće identifikaciju greške.
• Svaka klasa može da se nađe unutar posebne izvorne datoteke i nezavisna je od drugih klasa.
• 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.

Deo dijagrama hijerarhije klasa izuzetaka



Hijerarhija klasa izuzetaka

• 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).

System.Exception

• Izvedena iz klase System.Object; predstavlja baznu klasu za sve izuzetake

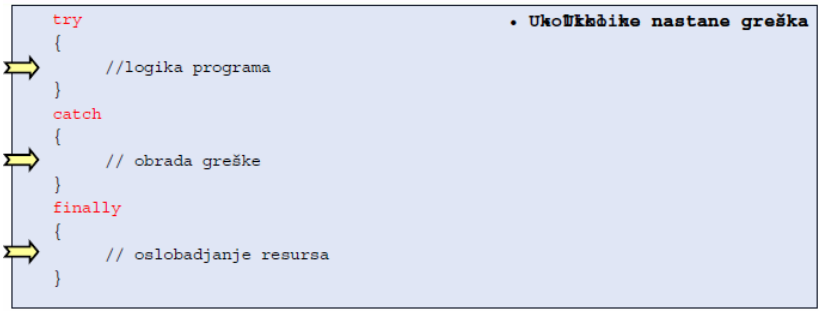
Svojstvo	Opis
HelpLink*	Sadrži link ka datoteci u kojoj su smeštene dodatne informacije o nastalom izuzetku.
InnerException*	Sadrži referencu na instancu izuzetka koji je izazvao tekući izuzetak.
Message*	Sadrži tekst koji opisuje uzrok nastanka izuzetka. Treba da potpuno opiše grešku i eventualno način njenog ispravljanja.
Source**	Sadrži naziv aplikacije ili objekta koji je izazvao izuzetak.
StackTrace**	Sadrži detalje o pozvanim metodama (na steku), kako bi se lakše pronašla metoda u kojoj je podignut izuzetak.
TargetSite**	Sadrži referencu na objekat kojim se opisuje metoda u kojoj je podignut izuzetak.

* Nisu obezbeđeni automatski, već se navode pre samog podizanja izuzetka
** Automatski obezbeđeni od strane izvršnog okruženja

Hvatanje izuzetaka

• Kako se sada ove klase izuzetaka koriste za otkrivanje grešaka?
• Program se deli u tri različita bloka:
• **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)
• Kako ovi blokovi hvataju greške?
• 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 (korak 5). Međutim ukoliko dođe do greške unutar **try** bloka izvršavanje se prenosi **catch** blok (sledeći korak).
• U **catch** bloku se vrši obrada greške.
• Na kraju **catch** bloka izvršavanje se automatski prenosi na **finally** blok.
• **Finally** blok se izvršava.

Sintaksa



• Blokovi catch i finally se mogu izostaviti.
• Naredbe break, continue i goto se mogu koristiti u finally bloku samo ukoliko ne prenose kontrolu izvan tog bloka, dok return naredba nikada ne bi trebala da se nađe u njemu.

Hvatanje izuzetaka
Kada nastane izuzetak prekida se normalno izvršavanje programa i traži se najbliži catch blok koji može da ga obradi (na osnovu njegovog tipa). Pretražuju se catch blokovi tekućeg try bloka po redosledu navođenja. Ako pretraga nije uspela prekida se izvršavanje tekuće metode i tok kontrole se prenosi na mesto njenog poziva. Pretraživanje se nastavlja na ovaj način sve dok se ne pronađe catch blok koji može da obradi nastali izuzetak ili ne stigne do kraja Main-a u kada će se prekinuti izvršavanje programa. Prekid izvršavanja programa se odvija kontrolisano. Ceo program je smešten unutar jednog velikog try bloka koji ima catch blok kojim se može obraditi bilo koji tip izuzetka. Kada se pronađe odgovarajući catch blok priprema se prenos kontrole na njegovu prvu naredbu. Pre samog prenosa izvršavaju se (po redosledu) svi finally blokovi povezani sa prethodno ispitivanim catch blokovima.

Podizanje izuzetaka
Ako u toku izvršavanja programa nastane greška "podiže se izuzetak", odnosno instancira se objekat odgovarajuće klase izuzetka i podiže se.

```
throw new klasaIzuzetka();
```

Izuzetci mogu biti podignuti na dva različita načina:
– **eksplicitno** navođenjem throw naredbe.
- momentalno i bezuslovno se podiže izuzetak
- izabrati onu klasu izuzetka koja u najboljoj meri opisuje nastalu grešku
- naredba koja sledi naredbu throw nikad neće biti izvršena.

– **implicitno** kada obrada naredbe ili izraza ne može normalno da se završi
- izvršno okruženje izvršava throw naredbu i podiže sistemski definisan izuzetak
- npr. slučaj celobrojnog deljenja kod koga je delilac 0 (System.DivideByZeroException)
- Dozvoljeno je podizanje izuzetaka u catch i finally bloku.

```
try {
if (b == 0) throw new DivideByZeroException(); // eksplicitno
...
int k = a / b; // implicitno
}
```

Catch blok
Svaki catch blok hvata određenu klasu izuzetaka.
Sintaksa catch bloka

```
catch ( tipIzuzetka identifikator )
{
// obrada izuzetka
}
```

Tip izuzetka mora biti ili System.Exception ili neki drugi tip koji je iz njega izveden (direktno ili indirektno).

Catch blok će obraditi izuzetke navedenog tipa i svih njegovih podtipova.
Identifikator, čije navođenje nije obavezno, je lokalna promenljiva čija se vrednost ne može promeniti.
Catch blok se izvršava samo u slučaju podizanja izuzetka.

Višestruki catch blokovi

Izvršno okruženje automastki hvata instancu izuzetka i prosleđuje je isključivo na osnovu njenog tipa odgovarajućem catch bloku.
S obzirom da postoji više različitih tipova izuzetaka, moguće je navesti i više catch blokova, pri čemu svaki od njih hvata i obrađuje određeni tip.

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

Navođenje višestrukih catch

- Redosled navođenja catch blokova je bitan.
  - izvršava se jedan catch blok i to prvi odgovarajući
  - navođenje se vrši od najspecifičnijeg izuzetka ka najopštijem izuzetku
  - ukoliko se navede prvo opštiji, a zatim specifičniji tip izuzetka prijaviće se greška
- Ne smeju se navesti dva catch bloka koji imaju isti tip izuzetka

```
try {...}

catch (ArgumentNullException izuzetak) {...}
catch (ArgumentException izuzetak) {...}

catch (ArgumentException izuzetak) {...}
catch (ArgumentNullException izuzetak) {...} // greška

catch (ArgumentException izuzetak) {...}
catch (ArgumentException izuzetak) {...} // greška
```

Opšti catch blok

- Catch blok čiji je tip izuzetka

System.Exception se naziva opšti catch blok.

  - može da obradi bilo koji izuzetak bez obzira na njegov tip, obzirom da su svi izuzeci izvedeni iz System.Exception klase.
  - nedostatak mu je što ne pruža informacije o prirodi greške
- Za jedan try blok
  - može se navesti samo jedan opšti catch blok
  - ako je naveden mora biti i poslednji
- Koristi se da bi se uhvatili izuzeci koji nisu obrađeni posebnim catch blokovima.

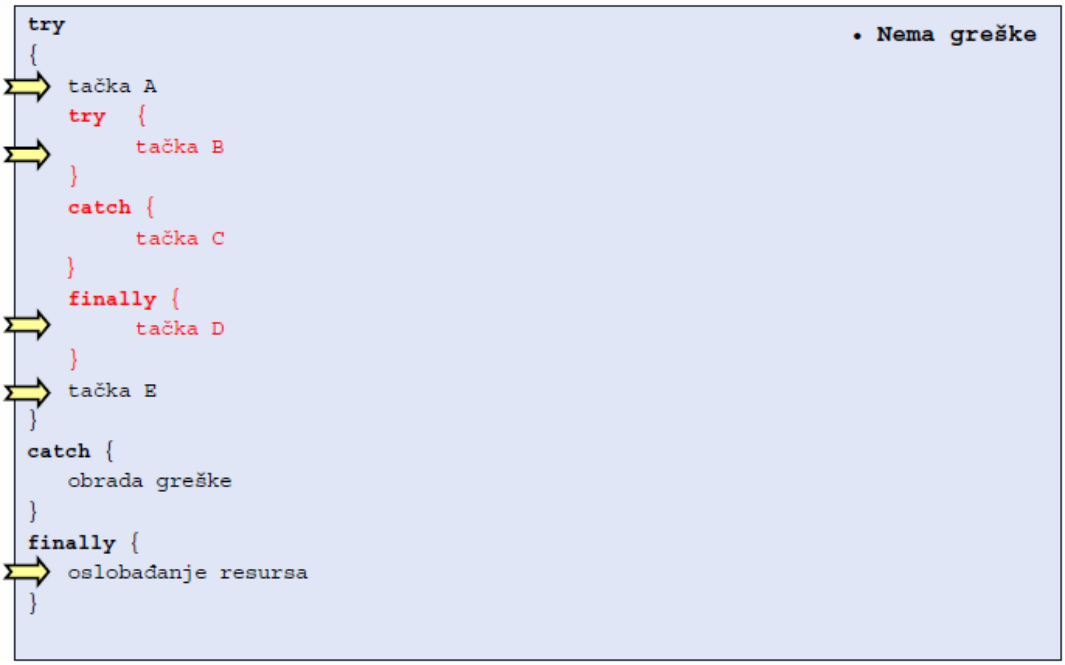
Sintaksa opšteg catch bloka

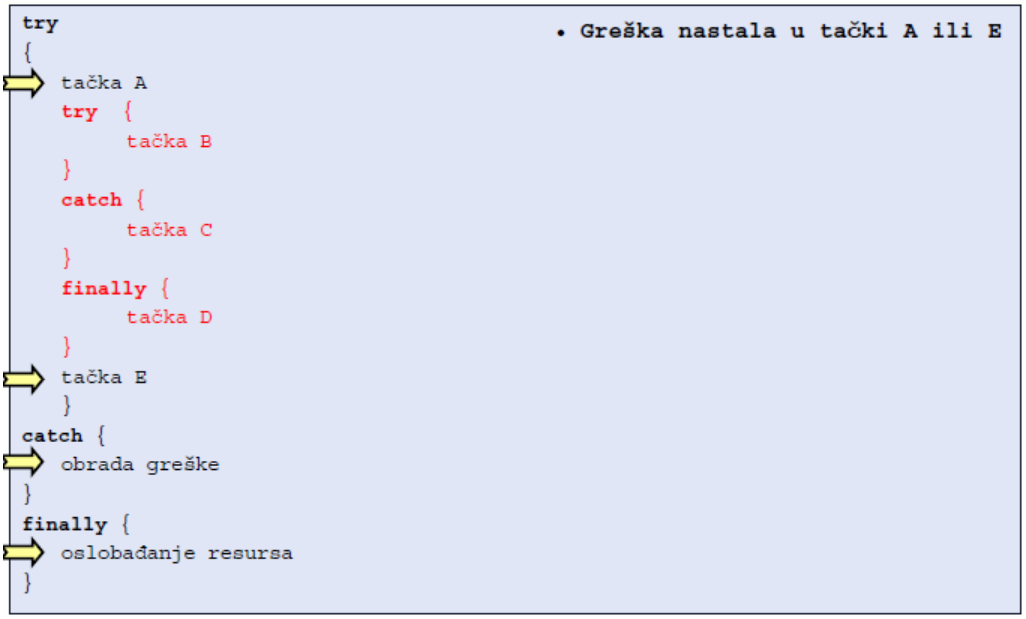
Opšti catch blok se može definisati na dva načina
`catch`

```
{
...
}
catch (System.Exception izuzetak)
{
...
}
```

Prvi je opštiji
• ne navodi se ni tip izuzetka ni identifikator (nema informacije o izuzetku).
• mogu se obraditi izuzeci koji nisu izvedeni iz System.Exception klase (kod napisan u drugom programskom jeziku - biblioteke).

Ugnježđeni try blokovi





```
try
{
   tačka A
   try  {
        tačka B
   }
   catch {
        tačka C
   }
   finally {
        tačka D
   }
   tačka E
}
catch {
   obrada greške
}
finally {
   oslobađanje resursa
}
```

- **Greška nastala u tački B**
  - postoji odgovarajući unutrašnji catch blok

```
try
{
   tačka A
   try  {
        tačka B
   }
   catch {
        tačka C
   }
   finally {
        tačka D
   }
   tačka E
}
catch {
   obrada greške
}
finally {
   oslobađanje resursa
}
```

- **Greška nastala u tački B**
  - ne postoji odgovarajući unutrašnji catch blok

```
try
{
   tačka A
   try  {
        tačka B
   }
   catch {
        tačka C
   }
   finally {
        tačka D
   }
   tačka E
}
catch {
   obrada greške
}
finally {
   oslobađanje resursa
}
```

- **Greška nastala u tački C**

Primena ugnježdenih try blokova

Modifikovanje tipa nastalog izuzetka

- kada originalni izuzetak ne opisuje nastali problem na adekvatan način
- u okviru catch bloka koji obrađuje originalni izuzetak podiže se novi izuzetak kojim se preciznije opisuje nastala greška.
- primer: ako je podignut izuzetak IOException, a poznat je kontekst u kome je nastao ovaj izuzetak jasno je da se radi o složenijem problemu – ne postoji tražena datoteka.

```
catch (IOException izuzetak)
{
throw new FileNotFoundException(ImeDatoteke);
}
```

Obrađivanje različitih tipova izuzetaka na različitim mestima u kodu

primer: ako imamo petlju u kojoj se mogu javiti raznoliki izuzeci od kojih neki mogu biti dovoljno ozbiljni da je potrebno iskočiti iz cele petlje, dok drugi mogu biti manje ozbiljni i jednostavno zahtevati da se iskoči iz iteracije.

try blok unutar petlje obrađuje manje ozbiljne greške, dok try blok koji obuhvata petlju obrađuje ozbiljnije greške.

Upotreba throw naredbe u catch bloku

Throw naredba se može koristiti unutar catch bloku za:

- ponovno podizanje tekućeg izuzetka (rethrow).
- unutar catch bloka, throw naredba se može navesti i bez ikakvog izraza.

sledeći blokovi imaju isti efekat:

```
catch (OutOfMemoryException izuzetak)
{ throw izuzetak; }
catch (OutOfMemoryException)
{ throw; }
```

podizanje novog izuzetka drugačijeg tipa:

```
catch (IOException izuzetak)
{ throw new FileNotFoundException(ImeDatoteke); }
```

u ovom slučaju se IOException instanca gubi

ukoliko su neophodne informacije koje ona sadrži izuzetak se prosleđuje kroz InnerException svojstvo novog izuzetka:

```
catch (IOException izuzetak)
{ throw new FileNotFoundException(ImeDatoteke, izuzetak); }
```

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.

Preporuke

Izbegavati korišćenje izuzetaka u normalnim i očekivanim situacijama
Nikada ne podizati izuzetke tipa System.Exception - nema informacija o specifičnosti nastale greške
Navoditi opis nastalog izuzetka

```
string opis = ...;
throw new FileNotFoundException(opis);
```

Podizati što specifičnije izuzetke - koristiti FileNotFoundException, umesto IOException
Nikada ne dozvoliti da izuzeci izađu iz Main-a neobrađeni, jedno od rešenja je da se stavi opšti catch blok u Main

```
static void Main() {
try {...}
catch (System.Exception izuzetak) {...}
}
```

Treba obraditi što je moguće više izuzetaka. Prilikom pisanja biblioteka ne treba obrađivati izuzetke.

Napomene

Ukoliko izuzetak nastane tokom izvršavanja destruktora i taj izuzetak nije obrađen, izvršavanje destruktora se prekida i destruktor osnovne klase (ukoliko postoji) se poziva.
Ukoliko ne postoji osnovna klasa (kao u slučaju tipa object) ili ako ne postoji destruktor osnovne klase izuzetak se odbacuje.