Osnove programskog jezika C#

Čas 3.

Imenski prostor ili prostor imena

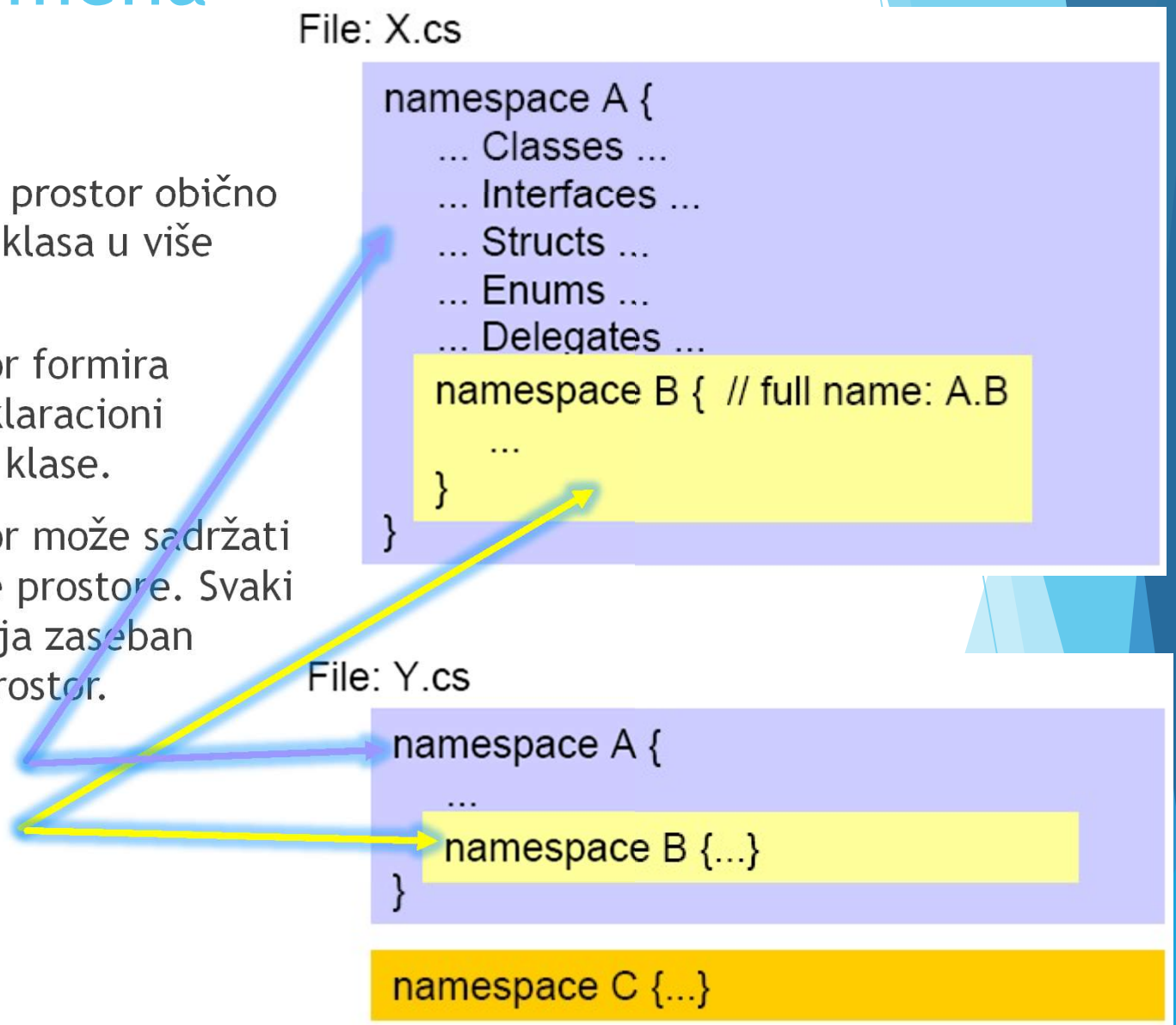
```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;


namespace WindowsFormsApplication2
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
    }
}
```

Imenski prostor

Obuhvata jednu ili više klasa i tačnije određuje naziv klase. Obično srodne klase pripadaju istom prostoru imena

Prostori imena

- Jedan imenski prostor obično obuhvata više klasa u više fajlova.
- Imenski prostor formira zajednički deklaracioni prostor za sve klase.
- Imenski prostor može sadržati druge imenske prostore. Svaki novi predstavlja zaseban deklarativni prostor.



Upotreba prostora imena

Color.cs

```
namespace Util {
    public enum Color {...}
}
```

Figures.cs

```
namespace Util.Figures {
    public class Rect {...}
    public class Circle {...}
}
```

Triangle.cs

```
namespace Util.Figures {
    public class Triangle {...}
}
```

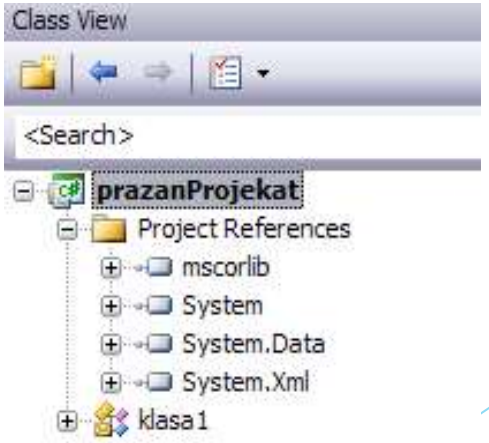
```
using Util.Figures;

class Test {
    Rect r;          // without qualification
    Triangle t;
    Util.Color c;    // with qualification
}
```

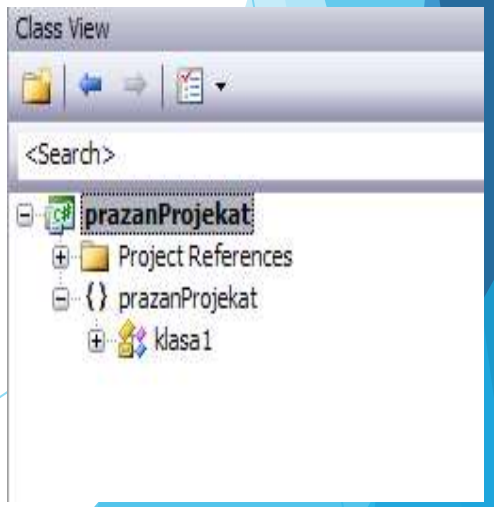
Napomena: Strani prostor imena mora biti uključen koristeći direktivu ***using ProstorImena*** ili se pri navođenju klase mora navesti cela putanja tj. puno ime klase.

Globalni (neimenovan) prostor imena

- Svaki C# program uključuje najmanje jedan, *globalan ili neimenovan* prostor imena (kao kod C++).
- Klase mogu imati isto ime ako ne pripadaju istom imenskom prostoru.
- Neimenovan prostor imena je prisutan ako ne postoji definisani:

```
//namespace prazanProjekat
//{
        class klasa1
        {
            public static void Main()
            {
                Console.WriteLine("Test");
            }
        }
//}
```



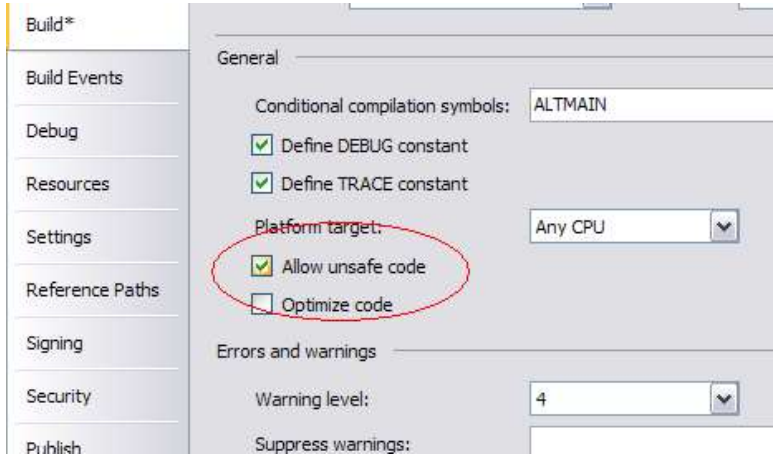


Nesiguran kod

- U C# uglavnom nema potrebe za korišćenjem pointera/pokazivača, osim...
  - Ako se poivaju funkcije C++ bibilioteke čiji su parametri pokazivači na promenljivu.
  - Upotreba pokazivača nekada povećava efikasnost.

unsafe

- Mehanizam korišćenja pokazivača u C# zahteva upotrebu rezervisane reči **unsafe**.
- Čest primer upotrebe pokazivača je pri kopiranju nizova – zbog veće brzine izvršavanja.
- Neophodno podešavanje prevodioca. Pogledati sliku ispot.




```
unsafe public static bool copyarr(int[] b, int[] a, int len)
{
    if (len > b.Length || len > a.Length)
    {
        return false;
    }
    fixed (int* A = a, B = b){
        int* pA = A;
        int* pB = B;
        for (int n = 0; n < len; n++)
            *pB++ = *pA++;
    }
    return true;
}
```

Statička polja i konstante

- Statička polja pripadaju klasi, ali ne i objektu.
- Na primer:

```
class Rectangle
{
    static Color defaultColor; // jedan po klasi
    static int scale;
    int x, y, width, height; // jedan po objektu
}
```

- Pristup iz iste klase:
  - `defaultColor...scale....`
- Pristup iz dugih klase:
  - `...Rectangle.defaultColor....Rectangle.scale....`

Statičke metode

```
class Rectangle
{
    static Color defaultColor; // jedan po klasi
    public static void ResetColor()
    {
        defaultColor = Color.White;
    }
}
```

- Pristup iz iste klase:
  - `ResetColor()`
- Pristup iz dugih klase:
  - `Rectangle.ResetColor()`

Objektno orijentisano projektovanje?

- Dizajniranje klase podrazumeva:
- 1. vizija ukupnog rešenja
- 2. planiranje, tj. ideje
  - 2.1. uočavanje klasa
    - Klasa Krug
    - Klasa Pravougaonik
- 3. definisanje osnovnih funkcija klase
  - Krug:
    - Čuva podatak o poluprečniku
    - Ima funkciju za računanje površine
- 3. definisanje osnovnih funkcija klase
  - Pravougaonik:
    - Čuva podatke o stranicama
    - Ima funkciju za računanje površine
- 4. prvo rešenje
  - “Korisnik ima dve poruke:
    - 1. Krug
    - 2. Pravougaonik”
- 4. jedan “scenario”
  - “Korisnik ima poruku:
    - 1. Krug
    - 2. Pravougaonik”
    - Izaberite opciju?
  - Pošto pritisne broj 1 ili 2 dobijaju se moguće poruke:
    - “Unesite poluprečnik: (krug)
    - Unesite stranicu a: (pravougaonik)
    - Unesite stranicu b: (pravougaonik)”
- 5. Po prijemu podataka, vrši se konstruisanje objekata klase Krug, Pravougaonik
- 6. Izvršava se metoda za računanje površine
- 7. Ispisuje se rezultat

Klasa

```
class Pravougaonik{
	private int a;
	private int b;
	public Pravougaounik(int x, int y){
		a = x;
		b = y;
	}
	public int P(){
		return a * b;
	}
}
```

```
class Krug{
	private int r;
	public Krug(int x)
	{
		r = x;
	}
	public double P()
	{
		return r*r*3.14;
	}
}
```

- Klasa definiše tip podataka koji se koristi u rešenju. Predstavlja specijalizovan tip podataka. Referentni. Elementi (instance) klase su objekti. Sve klase imaju jednu zajednicku klasu *object*

```
Pravougaonik p;
p = new Pravougaonik(12,21);
int povrsina = p.P();
```

```
Krug k;
k = new Krug(17);
double povrsina = k.P();
```

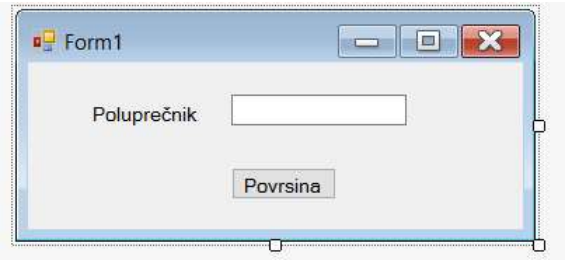
- Klasa je opis objekata koji se koriste u aplikaciji. Projektovanje klase znači osmišljavanje tipa podataka koji će omogućiti lako i efikasno rešavanje postavljenog zadatka, ali i naknadno održavanje napisang koda!

Osobine klase

- Klasa može da preuzme sve osobine neke druge klase i da joj se dodaju nove. Osobine su atributi (polja) i metode.
- Preuzimanje tih osobina se naziva “nasleđivanje” (u Javi “proširivanje”).
- Ceo kod mora biti u klasama.
  - Neke klase služe za pokretanje i rad aplikacije u os okruženju.
  - Druge za opis objekata koji ce se koristiti.

Primer:

Koncept:




```
private void button1_Click(object
sender, EventArgs e)
{
    // procitati uneti podatak

    // izracunati povrsinu

    // prikazati rezultat
}
```

Klasa:

```
class Krug
{
    public Krug()
    {
        r = 1;
    }
    public Krug(int r)
    {
        this.r = r;
    }
    public int r;
    public double povrsina()
    {
        double p = 0;
        p = r * r * Math.PI;
        return p;
    }
}
```

Rešenje:

```
private void button1_Click(object
sender, EventArgs e)
{
    // procitati uneti podatak
    string poluprecniStr =
    this.textBox1.Text;
    int r =
    int.Parse(poluprecniStr);

    // izracunati povrsinu
    Krug k = new Krug();
    k.r = r;
    double p = k.povrsina();

    // prikazati rezultat
    MessageBox.Show(p.ToString());
}
```

Nasleđivanje

```
class A // bazna/roditeljska/osnovna klasa
{
    int a;
    public A() { . . . } // konstruktor
    public void F() { . . . }
}
class B : A // izvedena/dete klasa
{
    int b;
    public B() { . . . }
    public void G() { . . . }
}
```

- B nasleđuje polje *a* i metodu *F()*, a dodaje svoju metodu *G()*.
  - *Konstruktori se ne nasledjuju na isti način, ali se pozivaju pri pozivu izvedenih konstruktora.*
  - *Nasleđene metode se mogu preklopiti (objašnjenje kasnije)*
- Klasa može da nasleđuje klasu, a ne može strukturu.
- Struktura ne može da nasleđuje neki drugi tip. Ona može samo da implementira (nasleđuje) interfejse.
- Klasa bez eksplicitno navedene roditeljske klase nasleđuje *Object*.

“B je A, ali A nije B”

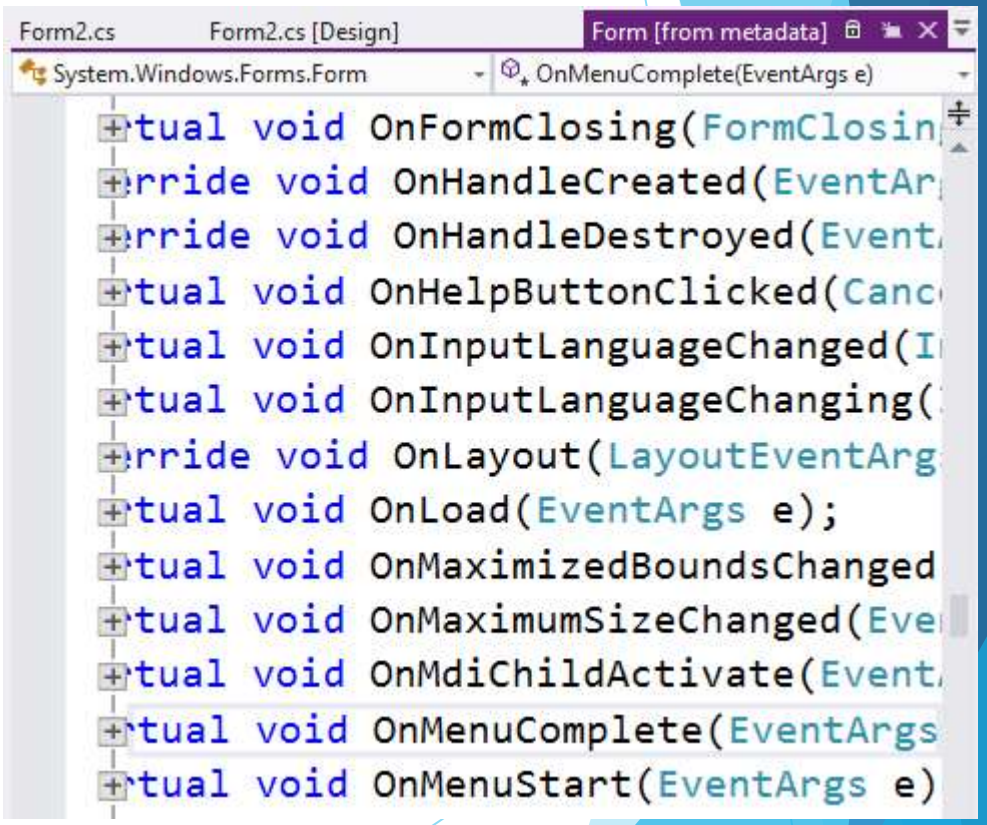
- Klasa A ima i koristi polja i metode svoje klase. Klasa B ima sve metode i polja klase A i klase B, ali može da koristi samo svoje i ono što je `public` ili `protected` u klasi A. Klasa B je istovremeno i klasa A. To znači da je objekat tipa B implicitno i objekat tipa A.
- `B b = new B();`
- `A a = new B(); // a je objekat koji ima sva obelezja obe klase ali je samo tipa A`
- Poziv metode iz klase A
- `B b = new B();`
- `b.F(); // ok`
- Konkretno, u klasi `Form` postoji metoda `OnLoad` koja se poziva pri učitavanju forme. Modifikator pristupa je `protected` pa se metoda može koristiti u izvedenim klasama.
- Dakle, u klasi `Form1` se može eksplicitno pozvati metoda `OnLoad.`
- `base.OnLoad(null);`

Međutim...

- Metoda `OnLoad` u roditeljskoj klasi `Form` se koristi na mnogo mesta, ali to ne sme da bude pod kontrolom programera koji koristi klasu `Form1` i tako se čuvaju osnovne funkcije postojeće klase.
- **Dakle, ako želimo da nešto uradimo pri učitavanju naših formi koje nastaju iz klase `Form1`, potrebno je da sve metode klase `Form1` koje su nasleđene od `Form` ne koriste više `OnLoad` metodu klase `Form`, već napisanu metodu iz klase `Form1`.**
- **OVO SE POSTIŽE KOMBINACIJOM**
- **virtual**
- **override**

Virtuelne metode i preklapanje

- Metoda `OnLoad` je definisana kao `virtuelna`.
- Ako se napiše metoda `OnLoad` koja je override onda se ostvaruje preklapanje postojeće metode sa novom metodom za novu klasu.
- Metoda koja je `override` mora imati iste privilegije kao i roditeljska (u konkretnom slučaju `protected`) i imati isti oblik kao roditeljska!



Objekat roditeljske klase

- Ključna reč `base` se koristi da bi se eksplicitno pozvala metoda ili polje roditeljske klase.

```
protected override void OnLoad(EventArgs e)
{
    base.OnLoad(e);
    this.Text = "drugi naslov";
}
```

- U konkretnom primeru, nova metoda radi isti posao kao i roditeljska uz dodatak vezan za promenu naslova forme.
- Zaključak: Za sve što se oslanjalo na `OnLoad` važi i dalje samo se za `Form1` koristi sopstvena metoda!!
- Napomena: U roditeljskoj klasi metoda `OnLoad` izaziva istoimeni događaj.

Preklapanje virtuelne metode

```
class A{
    public virtual void Klasa(){
        Console.WriteLine("tip class A");
    }
}
class B : A{
    public override void Klasa(){
        Console.WriteLine("tip class B");
    }
}
```

```
A a = new B();
a.Klasa(); // "tip class B"
```

```
void Obrada(A x)
{
    x.Klasa();
}
```

```
Obrada(new A()); // "tip class A"
Obrada(new B()); // "tip class B"
```

Modifikator *new*

- Elementi mogu biti deklarisani kao new u podklasi.
- Tako deklarisani skrivaju nasleđene elemente istog imena.

```
class A // bazna/roditeljska/osnovna klasa
{
    public int a;
    public A() { /*. . .*/ } // konstruktor
    public void F() { /*. . . */}
    public virtual void G() { /*. . . */}
}

class B : A // izvedena/dete klasa
{
    public new int a;
    public new void F() { /*. . .*/ }
    public new void G() { /*. . .*/ }
}
```

```
B b = new B();
b.a = 22;       // pristup B.a
b.F(); b.G();   // poziv B.F(), B.G()

((A)b).a = 22; // pristup A.a
((A)b).F(); ((A)b).G(); // A.F(), A.G()
```

Primer

```
class A {
    public virtual void F() {
        Console.WriteLine("I am A");
    }
}
class B : A {
    public override void F() {
        Console.WriteLine("I am B");
    }
}
class C : B {
    public new virtual void F() {
        Console.WriteLine("I am C");
    }
}
class D : C {
    public override void F() {
        Console.WriteLine("I am D");
    }
}
```

```
A a1 = new A();
a1.F(); // I am A
A b1 = new B();
b1.F(); //I am B
B b2 = new B();
b2.F(); // I am B
```

```
A c1 = new C();
c1.F(); // I am B
B c2 = new C();
c2.F(); // I am B
C c3 = new C();
c3.F(); // I am C

A d1 = new D();
d1.F(); // I am B
B d2 = new D();
d2.F(); // I am B
C d3 = new D();
d3.F();  // I am D
D d4 = new D();
d4.F(); // I am D
```

Nešto više o konstruktorima

Podrazumevani konstruktor klase

- Ako ne postoji deklarisan konstruktor u jednoj klasi, kompajler automatski generiše jedan konstruktor bez parametara - podrazumevani konstruktor.

```
class C { int x; }
C c = new C();      // ok
```

- Podrazumevani konstruktor inicijalizuje sva polja klase:
  - brojeve tipove na 0,
  - bool na false,
  - char na '\0',
  - reference na null
- Ako postoji deklarisan bar jedan konstruktor u klasi, podrazumevani se ne generiše!

```
class C {
    int x;
    public C(int y) { x = y; }
}

C c1 = new C();    // compilation error
C c2 = new C(3);   // ok
```

Podrazumevani konstruktor strukture

```
struct Complex
{
    double re, im;
    public Complex(double re, double im)
    {
        this.re = re;
        this.im = im;
    }
    public Complex(double re) : this(re, 0) { }
}
```

```
Complex c0;  // re = 0, im = 0
Complex c1 = new Complex();     // re = 0, im = 0
Complex c2 = new Complex(4);    // re = 4, im = 0
Complex c3 = new Complex(4,3); // re = 4, im = 3
```

- Za svaku strukturu kompajler generiše jedan podrazumevani konstruktor bez parametara, čak i da postoji neki drugi konstrukotr. Podrazumevani konstruktor postavlja sva polja strukture na nulu.
- Programer ne može (niti sme) da deklariše konstruktor bez parametara za strukturu.

Pozivanje konstruktora iz konstruktora iste klase

```
class Vozilo
{
    private string opis;
    private uint brojTockova;
    public Vozilo(string _opis, uint _brojTockova){
      opis = _opis;
      brojTockova = _brojTockova;
    }
    public Vozilo(string _opis) : this( _opis, 4 ){
      // dodatna inicijalizacija
    }
}
```

Konstruktori u kontekstu hijerarhije klasa

- U postupku pravljenja instance izvedene klase, angažovani su ne samo konstruktori te klase već i klasa koje su nasleđene – čak i kada se to ne naglasi:

```
public myClass(){
    Name = “no name”;
}
```

- Identično sa:

```
public myClass() : base (){
   Name = “no name”;
}
```

Primer sa parametrima:

```
public myClass(string name) : base (name)
{
    // ostatak inicijalizacije
    // specifican za myClass
}
```

Implicitno pozivanje konstruktora

Eksplic.

```
class A {
    ...
}

class B : A {
    public B(int x) {...}
}
```

```
B b = new B(3);
```

OK
- default constr. A()
- B(int x)

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

class B : A {
    public B(int x) {...}
}
```

```
B b = new B(3);
```

OK
- A()
- B(int x)

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

class B : A {
    public B(int x) {...}
}
```

```
B b = new B(3);
```

Error!
- no explicit call of the A() constructor
- default constr. A() does not exist

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

class B : A {
    public B(int x)
    : base(x) {...}
}
```

```
B b = new B(3);
```

OK
- A(int x)
- B(int x)

Statički konstruktor

Novina kod C# su statički besparametarski konstruktori:

```
class MyClass{
    static MyClass(){
        // kod za inicijalizaciju
    }
}
```

- IZVRŠAVA SE SAMO JEDANPUT
  - za razliku od drugih konstruktora koji se izvršavaju prilikom svakog pravljenja objekata iz date klase.
  - Korisni su za inicijalizaciju statičkih promenljivih.
  - U C++ ne postoji ništa analogno ovom.
- Vreme izvršavanja i redosled se ne garantuje, osim, naravno, da će biti pozvan pre instanciranja bilo kog objekta te klase!
- Modifikatori pristupa su besmisleni u slučaju ovih konstruktora jer ih poziva .NET runtime okruženje.
  - Zato nemaju parametre i može postojati samo jedan ovakav konstruktor klase

Destruktori

- Moguće ga je eksplicitno implementirati i on se može koristiti pri oslobađanju nekih nekontolisanih resursa. Može postojati samo jedan destruktor. Nema modifikator pristupa i argumente. Ne postoji za strukturu. Poziva ga GC.

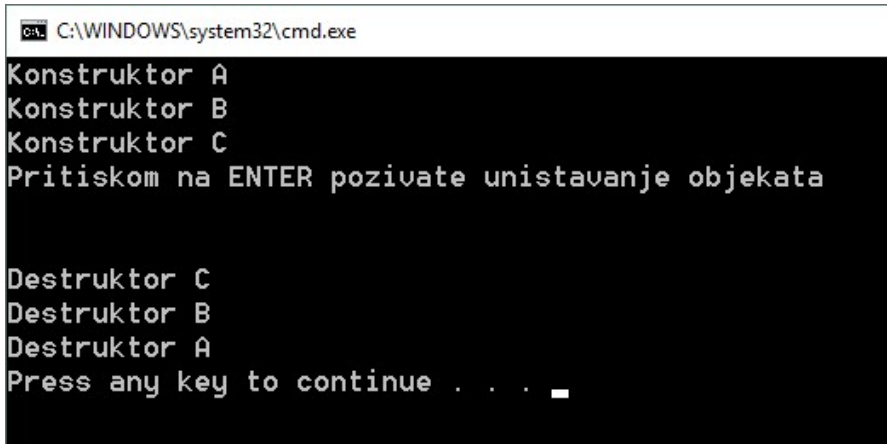
```
class Car {

      ~Car() // destructor {
             // cleanup statements...
          }
}
```

- Destruktor implicitno poziva `Finalize` metodu bazne klase. Zato je prethodni kod implicitno transformisan u sledeći kod:

```
protected override void Finalize() {
   try {
      // Cleanup statements...
   }
   finally{
      base.Finalize();
   }
}
```

```
class A{
    public A(){
        Console.WriteLine("Konstruktor A");
    }

    ~A(){
        Console.WriteLine("Destruktor A");
    }
}

class B : A{
    public B(){
        Console.WriteLine("Konstruktor B");
    }
    ~B(){
        Console.WriteLine("Destruktor B");
    }
}

class C : B{
    public C(){
        Console.WriteLine("Konstruktor C");
    }
    ~C(){
        Console.WriteLine("Destruktor C");
    }
}
```

```
C c = new C();
Console.WriteLine("Pritiskom na ENTER pozivate uništavanje objekata");
Console.ReadLine();
c = null;
Console.Read();
```



IDisposable

- Napomena:
- Oslobađanje zauzetih resursa se obično obavlja implementacijom metode `Dispose` interfejsa `IDisposable`
- Ovom metodom se zatvaraju ili oslobađaju neupravljivi(unmanaged) resursi kao na primer fajlovi, tokovi (streams). Ovaj metod, po dogovoru, se koristi za sve zadatke oslobađanja resursa.
- Metod `Despose` se poziva kada objekat više nije potreban, u kodu, ne od strane GC-a.

Zašto je implementacija kroz interfejs?

- Na taj način se garantuje propagacija oslobađanja kroz hijerarhiju svih klasa.
- PRIMER:
  - A sadrži objekat B, i objekat B sadrži neki objekat C, onda A-**Dispose** mora da pozove B-**Dispose** koji zatim poziva C-**Dispose**. Objeat mora pozvati **Dispose** metod bazne klase ako je u baznoj klasi implementiran IDisposable.

Primer:

```
public class MyClass : IDisposable{
        public void Dispose() // implementacija metode iz interfejsa IDisposable
        {
                Dispose(true); // preopterecivanje metoda Dispose
                GC.SuppressFinalize(this);
        }
        private void Dispose( bool disposing){
                If(disposing)
                {
                        // ciscenje koda
                }
        }
        ~MyClass(){
                Dispose(false);
        }
}
```

Operator “is” (provera tipa pod.)

```
class A {...}
class B : A {...}
class C: B {...}
```

Assignments

```
A a = new A();      // static type of a: the type specified in the declaration (here A)
                    // dynamic type of a: the type of the object in a (here also A)
a = new B();        // dynamic type of a is B
a = new C();        // dynamic type of a is C

B b = a;            // forbidden; compilation error
```

Run time type checks

```
a = new C();
if (a is C) ...     // true, if dynamic type of a is C or a subclass; otherwise false
if (a is B) ...     // true
if (a is A) ...     // true, but warning because it makes no sense

a = null;
if (a is C) ...     // false: if a == null, a is T always returns false
```

Primer upotrebe operatora “is”

```
1.  class Class1{}
2.  class Class2{}

3.  class IsTest{
4.      static void Test(object o) {
5.          Class1 a;
6.          Class2 b;

7.          if (o is Class1) {
8.              Console.WriteLine("o je Class1");
9.              a = (Class1)o;
10.             // Do something with "a."
11.         }
12.         else if (o is Class2){
13.             Console.WriteLine("o je Class2");
14.             b = (Class2)o;
15.             // Do something with "b."
16.         }
17.         else{
18.              Console.WriteLine("o nije Cl1 ni Cl2.");
19.         }
20.     }
```

```
1.  static void Main()
2.      {
3.          Class1 c1 = new Class1();
4.          Class2 c2 = new Class2();
5.          Test(c1);
6.          Test(c2);
7.          Test("a string");
8.      }
9.  }
```

Operator “as”

```
// "cast" (kastovaje)
A a = new C();
B b = (B)a;
C c = (C)a;

a = null;
c = (C)a;
```

- **as** operator je kao operator konverzije tipova (cast) osim što daje vrednost null ako konverzija nije moguća umesto da izbacuje izuzetak.

```
// as
A a = new C();
B b = a as B;
C c = a as C;

a = null;
c = a as C; // c==null
```

Klasa *Object*

- Predstavlja koren svih klasa u .NET
- Zapazite da se konstruktor ove klase poziva iz konstruktora svih izvedenih klasa, ali takođe može biti pozivan direktno pri kreiranju neke instance ove klase.
- Metode

  `Object.Equals(Object)`,
  `Object.Equals(Object, Object)`

  Određuje da li je tekući objekat jednak sa objektom koji je argumenat metode.

  `public virtual string ToString()`,
  Vraća `string` koji predstavlja tekući objekat `Object`.

  `protected virtual void Finalize()`
  Obavlja operaciju čišćenja resursa koje drži tekući objekat koji se uništava.

```
int x = 123;
float y = (float)22.44;
bool b = true;
object o = x;
Console.WriteLine("o=" + o.ToString() );
o = y;
Console.WriteLine("o=" + o.ToString());
o = b;
Console.WriteLine("o=" + o.ToString());
```

```
class student{
    public string ime;
    public string prezime;
    public student(string i, string p){
        ime = i;
        prezime = p;
    }
    override public string ToString(){
        return prezime + " " + ime
    }
}

student s1 = new student("aca", "peric");
o = s1;
Console.WriteLine("o=" + o.ToString());
```

typeof, sizeof

- `typeof` – vraća objekat tipa `Type` kojim je opisan tip podataka.

```
Type t = typeof(int);
Console.WriteLine(t.Name);     // ➔ Int32
```

- `sizeof` – vraća veličinu tipa u bajtovima, samo za predefinisane vrednosne tipove

```
int size1 = sizeof(int);
int size2 = 0; // sizeof(int[]);
int size3 = 0; // sizeof(string);
int size4 = 0; // sizeof(IntPtr);
int size5 = sizeof(decimal);
int size6 = sizeof(char);
int size7 = sizeof(bool);
int size8 = sizeof(byte);
int size9 = sizeof(Int16); // Equal to short
int size10 = sizeof(Int32); // Equal to int
int size11 = sizeof(Int64); // Equal to long
```

Output

```
4
0
0
0
16
2
1
1
2
4
8
```

Aplikacije u grafičkom okruženju

- Kako od konzolne aplikacije do `Windows` aplikacije?
  - Nova referenca u projektu za korscenje `Windows.Forms` biblioteke
- Korišćenje projekta tipa Windows application.
- Dodati klasu `Figura`;
- Dodati klasu `Krug` i `Pravougaonik`;
- Primeniti `as` i `is` operatore.