

# C# 02. Klase, interfejsi, svojstva, operatorske funkcije, indekseri

Prof. dr Suzana Stojković

Dr Martin Jovanović

Dipl. inž. Ivica Marković

Dipl. inž. Teodora Đorđević

# Sadržaj

- ▶ Prostori imena - namespaces
- ▶ Modifikatori pristupa
- ▶ Klase
- ▶ Nasleđivanje
- ▶ Apstraktne metode i klase
- ▶ Interfejsi
- ▶ Prenos parametara
- ▶ Svojstva (properties)
- ▶ Indekseri
- ▶ Operatori
- ▶ Parcijalne klase
- ▶ Klasa Object

# Prostori imena - *namespaces*

- ▶ Ključna reč **namespace** deklarise skup povezanih objekata
  - ▶ Slično kao **package** u Javi
- ▶ Prostor imena može da se uključi u naš program ključnom rečju **using**
  - ▶ Slično kao **import** u Javi
  - ▶ Razlika u odnosu na Javu:  

```
using System; // u C#u uključuje sve klase iz prostora imena System, ne i iz
                // njegovih potprostora imena npr. System.Text
```

```
import system.*; // ekvivalent u Javi, mora da se doda tačan naziv klase
                // ili zvezdica da bi se uključile sve klase
```
- ▶ Umesto uključivanja prostora imen sa **using** može da se koristi i tzv. kvalifikovano ime objekta oblika *NazivProstoraImena.NazivObjekta*
  - ▶ Npr. `System.String`, `System.Int32`, `Namespace1.Namespace2.NekaKlasa`

# Prostori imena - *namespaces*

## ► Primer - ključna reč using

### ► `using System;`

```
using System.Text;
namespace MojProstorImena
{
    class Program
    {
        static void Main(string[] args)
        {
            String s = "Zdravo";
            ...
        }
    }
}
```

## ► Primer - korišćenje kvalifikovanog imena

### ► `namespace MojProstorImena`

```
{
    class Program
    {
        static void Main(string[] args)
        {
            System.String s = "Zdravo";
            ...
        }
    }
}
```

# Modifikatori pristupa

- ▶ Postoje **public**, **protected**, **private** sa istim značenjem kao u C++u i Javi i dodat je **internal** i kombinacija **protected internal**
- ▶ **public** - pristup je moguć svuda
- ▶ **protected** - pristup je moguć samo u okviru tekuće klase ili u okviru klasa izvedenih iz tekuće klase
- ▶ **internal** - pristup u okviru tekućeg programa (**assembly**)
- ▶ **protected internal** - pristup u okviru tekućeg programa (**assembly**) ili u izvedenim klasama (koje ne mora da budu u okviru istog programa)
- ▶ **private** - pristup samo u okviru tekuće klase

# Modifikatori pristupa

- ▶ Članovi klasa (atributi, metode, *properties*, tipovi unutar klasa)
  - ▶ Mogu da imaju bilo koji od navedenih 5 nivoa pristupa
  - ▶ Podrazumevani nivo pristupa **private** ako se ne navede modifikator pristupa (slično C++u, različito od Jave)
- ▶ Članovi struktura (atributi, metode, *properties*, tipovi unutar struktura)
  - ▶ Mogu da imaju public, internal ili private nivo pristupa
  - ▶ Ne postoje **protected** ni **protected internal** jer nema nasleđivanja struktura
  - ▶ Podrazumevani nivo pristupa **private** ako se ne navede modifikator pristupa (slično C++u, različito od Jave)

# Modifikatori pristupa

- ▶ Tipovi (klase, strukture, enumeracije)
  - ▶ Mogu da imaju nivo pristupa public ili internal
  - ▶ Podrazumevani nivo pristupa je internal ako se ne navede modifikator pristupa
- ▶ Prostori imena - *namespaces*
  - ▶ Ne mogu da imaju modifikatore pristupa
  - ▶ Podrazumevani nivo pristupa public
- ▶ Članovi enumeracija ili interfejsa
  - ▶ Ne mogu da imaju modifikatore pristupa
  - ▶ Podrazumevani nivo pristupa public

# Assembly

- ▶ Na prethodnim stranicama pojam **assembly** smo prevodili kao program, ali preciznije značenje u .NET-u je malo drugačije.
- ▶ Engleska imenica assembly može da ima više značenja - sklop nekih povezanih elemenata, skupština, zbor. Glagol assembly znači povezivati, sklapati (npr. neke jednostavnije elemente u neki složeniji mehanizam).
- ▶ U .NET-u **assembly** je osnovna jedinica programa koju možemo negde da instaliramo, da joj dodelimo verziju, da ograničimo prava pristupa toj jedinici za različite korisnike...
- ▶ .NET **assembly** je rezultat kompajliranja jednog projekta iz okruženja Visual Studio i sadrži kompletan kod iz tog projekta (u kompajliranom MSIL formatu) i druge resurse (bitmape, ikonice, tabele stringova sa prevodima aplikacije na različite jezike itd.).

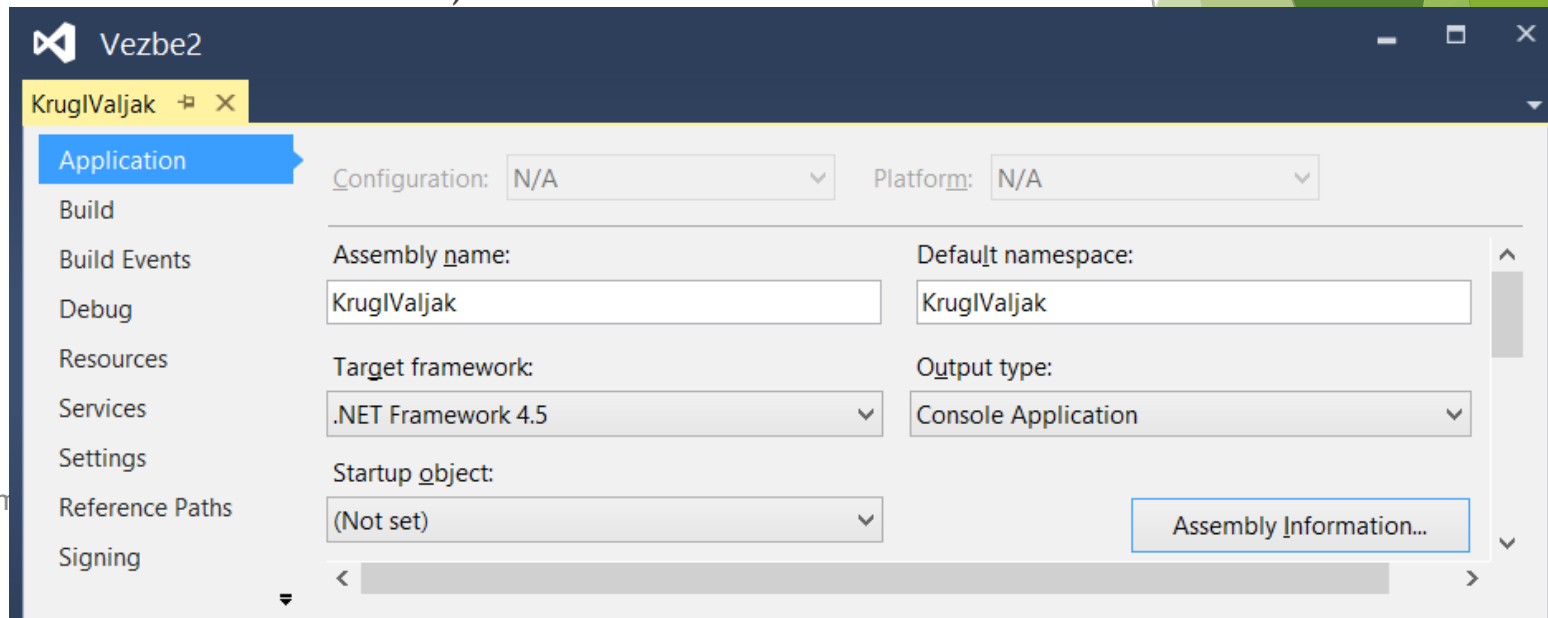
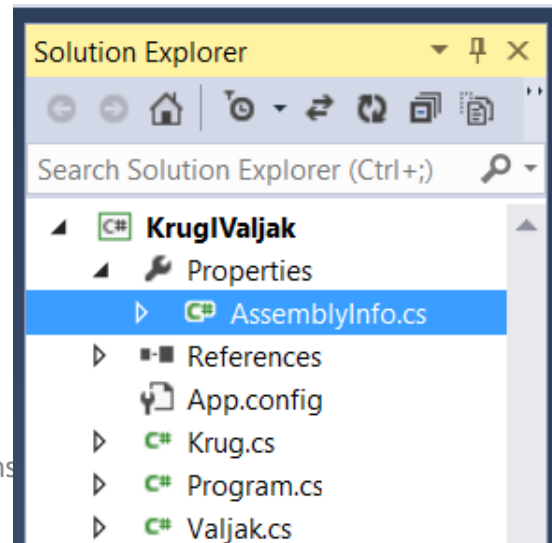


# Assembly

- ▶ .NET **assembly** se obično čuva u jednom fajlu koji se smešta u projektnom folderu u podfolderu “bin” i taj fajl može da ima ekstenziju .EXE ili .DLL:
  - Ekstenzija tog fajla je .EXE ako **assembly** predstavlja konzolnu aplikaciju (do sada smo obradili samo takve aplikacije) ili Windows aplikaciju (ove aplikacije će biti obrađene pri kraju kursa);
  - Ekstenzija tog fajla je .DLL ako **assembly** predstavlja biblioteku klasa (skraćenica od *Dynamic-link library*) - .DLL **assembly** ne možemo direktno da izvršimo kao samostalnu aplikaciju već se klase sačuvane u DLL fajlu referenciraju i koriste iz izvršnih **assembly**-ja (DLL projekti i referenciranje jednog projekta u okviru drugog biće obrađeno detaljnije na narednim časovima).

# Assembly

- ▶ Svaki .NET projekat mora da sadrži u folderu **Properties** fajl pod nazivom **AssemblyInfo.cs** i tu se nalaze bitna podešavanja za assembly koji projekat generiše (kao na slici dole levo, primer projekta KruglValjak).
- ▶ Neka od podešavanja projekta dostupna su na “Project Properties” pogledu do koga možemo da dođemo desnim klikom na naziv projekta u “Solution Explorer” pogledu pa izborom opcije “Properties”(kao na slici dole desno).



# Klase

- ▶ Ključna reč **class** isto kao u C++u ili Javi
- ▶ Za razliku od C++a ne postoje heder fajlovi
- ▶ Slično kao u Javi deklaracija i implementacija klase nisu odvojene
- ▶ Postoji mogućnost da implementacija klase bude podeljena u više fajlova (detaljnije objašnjenje sledi kasnije)
- ▶ Primer implementacije klase: projekat KruglValjak, fajl Krug.cs

# Klase - članovi tipa (*static*)

- ▶ Atributi i metode se definišu identično kao i u Javi
- ▶ Postoji ključna reč **static** koja ima isto značenje kao i u Javi
  - Ispred naziva atributa označava da taj atribut pripada tipu tj. klasi, a ne nekoj od instanci klase (postoji samo jedna kopija tog atributa u memoriji)
  - Ispred naziva metode označava da ta metoda pripada tipu tj. klasi, a ne nekoj od instanci klase (**static** metoda nema pristup atributima koji nisu označeni sa **static**)
- ▶ Pristup statičkim članovima klase:
  - Izvan te klase `<ImeKlase>.<ImeČlana>`
  - Unutar te klase `<ImeKlase>.<ImeČlana>` ili samo `<ImeČlana>`.

# Klase - *readonly* i *const* atributi

- ▶ Postoji ključna reč **readonly** koja ima isto značenje kao i upotrebu **final** kod atributa u Javi.
- ▶ **Readonly** atribut je promenljiva kojoj mora da se dodeli vrednost jednom (pri deklaraciji ili u konstruktoru) i ne može više nikada da se promeni.
- ▶ **Readonly** atribut može da se deklarise kao **static** (**static readonly**) i kao **non-static** (bez ikakvog dodatnog modifikatora).
- ▶ Postoji ključna reč **const** (za razliku od Java) za označavanje konstante klase.
- ▶ Konstanti se kod deklaracije mora dodeliti vrednost i ne može se kasnije menjati.
- ▶ Primer: **public const float PI = 3.14159f;**

# Klase - *readonly* i *const* atributi

- ▶ Razlika između **static readonly** i **const** atributa:
  - ▶ **static readonly** je promenljiva smeštena u nekoj memorijskoj lokaciji čija vrednost se uzima i koristi u toku izvršenja programa;
  - ▶ **const** je konstanta čije simboličko ime se na mestu korišćenja u kodu već u fazi kompajliranja menja direktno vrednošću te konstante.
- ▶ Sličnosti između **static readonly** i **const** atributa:
  - ▶ Postoji samo jedna kopija na nivou tipa (klase) u oba slučaja
  - ▶ I **static readonly** promenljivoj i konstanti klase se pristupa sa `<ImeKlase>.<ImeAtributa>`

# Klase - Konstruktori

- ▶ Osnovna sintaksa je ista kao u C++ ili Javi

- ▶ `public Krug(float r)`

- `{ this.r = r; }`

- ▶ Poziv drugog konstruktora iste klase je isti kao u C++u - koriste se `:` i ključna reč `this`, za razliku od Jave gde se `this` poziva kao prva naredba u konstruktoru

- ▶ `public Krug()`

- `: this(0.0f)`

- `{`

- `}`

# Nasleđivanje

- ▶ Moguće je samo jednostruko nasleđivanje kao u Javi
- ▶ Ne postoji ključna reč **extends** za razliku od Jave već se koristi operator **:** kao u C++u
- ▶ Ne mogu se dodati modifikatori pristupa kod nasleđivanja za razliku od C++a, podrazumeva se **public** nasleđivanje
- ▶ Primer: projekat KruglValjak, fajl Valjak.cs
  - ▶ 

```
class Valjak : Krug
```

```
{
```

```
...
```



# Nasleđivanje - Ključna reč *base*

- ▶ Koristi se za pristup osnovnoj klasi iz njene izvedene klase, slično kao ključna reč **super** u Javi
- ▶ Poziv konstruktora osnovne klase iz konstruktora izvedene klase - slično C++u koristi se operator `:` ali se umesto naziva osnovne klase navodi ključna reč **base**
  - ▶ 

```
public Valjak(float r, float h)  
    : base(r)  
{ this.h = h; }
```
- ▶ Poziv metode osnovne klase u slučaju kada je ona predefinisana u izvedenoj klasi, a želimo implementaciju baš iz osnovne klase
  - ▶ 

```
public float Zapremina()  
{ return base.Povrsina() * h; } // Treba nam površina kruga za računanje zapremine  
    // valjka, bez ključne reči base Povrsina() vraća površinu valjka.
```

# Nasleđivanje - Virtualne metode

- ▶ Metode **podrazumevano nisu virtualne**, slično kao u C++u, a za razliku od Jave gde su sve metode virtualne
- ▶ Ako se u osnovnoj klasi Krug metoda Povrsina() ne označi kao virtualna, a izvrši se sledeći kod rezultat će biti pogrešan
  - ▶ `Krug v = new Valjak(1, 1);`  
`Console.WriteLine("Površina valjka: " + v.Povrsina());`
- ▶ Pozvaće se metoda Povrsina() iz osnovne klase Krug na osnovu deklaracije promenljive v, a neće se vršiti provera stvarnog tipa promenljive v u trenutku poziva u vreme izvršenja jer metoda Povrsina nije virtualna

# Nasleđivanje - Virtualne metode

- ▶ Za rešenje prethodnog problema metoda Povrsina() u osnovnoj klasi Krug se označava kao virtualna
  - ▶ U klasi Krug

```
public virtual float Povrsina()
{ return r * r * Krug.PI; }
```
- ▶ Da bi metoda izvedene klase preklopila metodu osnovne klase sa istim potpisom **obavezno je korišćenje ključne reči `override`**
  - ▶ U klasi Valjak
  - ▶ 

```
public override float Povrsina()
{ return base.Povrsina() * 2 + 2 * r * h * Krug.PI; }
```
- ▶ Bez ključne reči **`override`** u metodi izvedene klase, kompajler će ovu metodu posmatrati kao potpuno novu, a ne kao metodu koja virtualno preklapa metodu osnovne klase
- ▶ Virtualne metode **ne mogu biti `private`**, jer se virtualno ponašanje pokazuje tek van klase

# Apstraktne metode i klase

- ▶ Identično kao u Javi apstraktne klase su klase koje sadrže ključnu reč **abstract** u deklaraciji
- ▶ Ne mogu se instancirati, samo je moguće iz njih izvesti druge klase
- ▶ Identično kao u Javi apstraktne metode su metode koje sadrže ključnu reč **abstract** u deklaraciji i nemaju implementaciju (telo metode)
- ▶ Apstraktne metode mogu da postoje samo u okviru apstraktne klase
- ▶ Za predefinisanje apstraktne metode u izvedenoj klasi važi isti princip kao i za predefinisanje virtualne metode - obavezna je ključna reč **override** i isti modifikator pristupa kao kod apstraktne metode u osnovnoj klasi
- ▶ Apstraktne metode **ne mogu** biti **private**, slično kao virtualne metode
- ▶ **Primer: projekat Brojevi**

# Nasleđivanje - Ključna reč **sealed**

- ▶ Ako smo klasu označili kao **sealed** iz nje nije moguće izvesti nijednu drugu klasu (u Javi se za to koristi ključna reč **final** ispred naziva klase)
- ▶ Primer:
  - ▶ **sealed** class Krug
  - ▶ {
  - ▶     protected float r;
  - ▶     ...
  - ▶ }
- ▶ Najpoznatiji primer iz .NET biblioteke klasa: **System.String**

# Interfejsi

- ▶ Isto kao i u Javi ključna reč **interface**
- ▶ Interfejsom se zadaju samo metode koje klasa treba da implementira, ne i kako treba da ih implementira
- ▶ Podrazumeva se da su sve metode interfejsa javne i apstraktne, ali ukoliko to pokušamo da eksplicitno zadamo **public** ili **abstract** kompajler će prijaviti **grešku** (za razliku od Jave)
- ▶ Isto kao i u Javi klasa može da implementira jedan ili više interfejsa
- ▶ Ne postoji ključna reč **implements** za razliku od Jave već se interfejsi koje klasa implementira navode iza operatora **:** međusobno razdvojeni zarezom ako ih ima više
- ▶ **Primer: projekat Iterator**

# Metode i prenos parametara

- ▶ Po „default-u“ prenos parametara kod poziva funkcija se radi po vrednosti tj. u pozvanoj funkciji postoji samo kopija promenljive iz pozivajuće funkcije:
  - ▶ Kod vrednosnih tipova prenose se kopije vrednosti
  - ▶ Kod referentnih tipova prenose se kopije referenci na objekte u dinamičkoj zoni memorije (kopija reference i dalje pokazuje na originalni objekat)
- ▶ Postoji ključna reč **ref** kojom zadajemo kompajleru da ne radi sa kopijama promenljivih iz pozivajuće funkcije već sa samim promenljivama iz pozivajuće funkcije
  - ▶ Radi isto kao operator **&** ispred argumenta funkcije u C++u
- ▶ Detaljnije objašnjenje u primeru: projekat PrenosParametara

# Metode i prenos parametara

- ▶ Ključna reč **out** ispred argumenta funkcije kaže kompajleru da je ta promenljiva istovremeno i povratna vrednost funkcije pa na taj način funkcija može imati više od jedne povratne vrednosti
- ▶ "Out" promenljivoj se obavezno dodeljuje vrednost u telu pozvane funkcije, inače kompajler prijavljuje grešku
- ▶ Zbog toga je dozvoljeno da "out" promenljiva bude neinicijalizovana prilikom poziva funkcije
  
- ▶ Ključna reč **params** može da stoji samo uz promenljivu tipa niz
- ▶ Omogućava nam da funkciju pozovemo sa proizvoljnim brojem argumenata od kojih je svaki istog tipa kao i pojedinačni elementi niza
- ▶ Detaljnije objašnjenje u primeru: projekat PrenosParametara



# Svojstva (Properties)

- ▶ Jezička konstrukcija koja menja *getter* i *setter* metode iz Java
- ▶ Primer Java klase za predstavljanje tačke u 2D prostoru:

```
▶ public class Tacka {  
    private double x, y;  
    public double getX() {  
        return x;  
    }  
    public void setX(double value) {  
        this.x = value;  
    }  
    ...  
}
```

# Svojstva (Properties)

Idetifikator koji označava naziv svojstva

- ▶ Omogućavaju sličnu funkcionalnost, samo bez eksplicitnog pisanja metoda
- ▶ Prethodni primer u C#-u:

- ▶ public class Tacka

{

private double x, y;

public double X // property, po konvenciji se naziv piše velikim početnim slovom

{

**get** { return this.x; } // get deo property-ja, kod ekvivalentan getter metodi u Javi

**set** { this.x = **value**; } // set deo property-ja, kod ekvivalentan setter metodi u Javi

}

// ključna reč value označava promenljivu istog tipa kao i property

...

// čija vrednost se dodeljuje property-ju

Ključna reč **get**

Ključna reč **set**

Ključna reč **value**

# Svojstva (Properties)

- ▶ Ključne reči:
  - **get** - označava get deo svojstva, interno će se prevesti u metodu bez argumenata, a istog povratnog tipa kao što je tip celog svojstva pa zbog toga mora da vraća objekat tog tipa
  - **set** - označava set deo svojstva, interno će se prevesti u metodu *void* povratnog tipa, a sa jednim argumentom istog tipa kao što je tip celog svojstva i naziva **value**
  - **value** - naziv promenljive, argumenta set dela svojstva
- ▶ Formalno svojstvo (property) liči na atribut - kod deklaracije i korišćenja ne postoje zagrade () za poziv metode
- ▶ U stvari se svojstvo ponaša identično kao metoda

# Svojstva (Properties)

- ▶ Kompajler svojstvo (property) prevodi interno u get i set metodu - slično onome što sami pišemo u Javi.
- ▶ Programer nema kontrolu nad tim interno generisanim metodama - vidi samo kod u svojstvu (property-ju) koji je napisao.
- ▶ Svojstvo (property) može da sadrži proizvoljan kod - kao što može i metoda
- ▶ Svojstvo (property) može da bude statičko, virtualno, apstraktno, član interfejsa - kao što može i metoda (**kao primer videti projekat IteratorProperties**):

```
▶ interface ITacka
{
    double X { get; set; }
    double Y { get; }
}
```

# Svojstva (Properties)

- Primer korišćenja svojstava iz ranije navedene klase Tacka u npr. Main metodi:

...

```
Tacka t = new Tacka();
```

```
t.X = 1.5; // Dodeljujemo vrednost svojstvu X. Interno se ovaj kod prevodi u  
// set deo svojstva. Argument value iz setera uzima vrednost sa desne strane  
// znaka dodele (1.5). Seter jedino možemo da pozovemo na ovaj način - tako  
// što izvršavamo dodelu neke vrednosti svojstvu.
```

```
Console.WriteLine(t.X); // Kada ovako neka funkcija pokušava da pročita  
// vrednost svojstva, izvršiće se get deo koji vraća vrednost odgovarajućeg tipa.  
double x = t.X; // I u ovoj situaciji se poziva get deo svojstva.
```

...

# Svojstva (Properties)

- ▶ Zavisno od toga koje delove sadrže postoje 3 vrste svojstava:
  - ▶ *read-only*: imaju samo get deo
  - ▶ *write-only*: imaju samo set deo
  - ▶ *read-write*: imaju i get i set deo
- ▶ **Automatsko svojstvo** - ne pišemo kod za get i set deo pa kompajler **automatski interno generiše odgovarajući atribut** i povezuje ga sa svojstvom.
- ▶ U ovom slučaju ne znamo kako se zove taj atribut i nećemo moći da mu pristupimo (osim preko svojstva).
- ▶ **Sintaksa je slična** kao kod apstraktnih svojstava ili svojstava članova interfejsa, ali je **značenje drugačije**:

```
public class Tacka {  
    public double X { get; set; }  
}
```

# Svojstva (Properties)

- Uobičajena greška kod pisanja svojstava je da umesto naziva atributa iskoristimo naziv svojstva (najčešće se ova dva naziva razlikuju samo u veličini slova)

```
public class Tacka {  
    private double x, y;  
    public double X {  
        get { return this.X; } // prva greška, ispravno je malo x kao naziv atributa  
        set { this.X = value; } // druga greška, ispravno je malo x kao naziv atributa  
    }  
}
```

- Pošto se i getter i setter prevode u metode, greške označene crvenom bojom u prethodnom primeru predstavljaju rekurzivni poziv te iste metode. Kompajler ovo neće da prepozna kao grešku, ali će u toku izvršenja da dođe do izuzetka (*StackOverflow*) i prekida rada programa.
- **Primer: projekat StudentiLOcene**

# Indekseri

- ▶ U C++u postoji preklapanje operatora [] čime možemo zadati da se naša proizvoljna klasa ponaša kao niz/matrica
- ▶ U C#u operator [] se ne može preklopiti
- ▶ Umesto toga se koriste indekseri
- ▶ Indekseri su posebna vrsta svojstava (*property*):

```
class Niz {  
    private int[] brojevi;  
    public int this[int i] {  
        get {return brojevi[i];}  
        set {brojevi[i] = value;}  
    }  
    ...  
}
```

Indekser nema svoj identifikator, zadaje se ključnom rečju **this** iza koje slede []

Argument indeksera **int i** se zadaje unutar zagrada []

Argument indeksera **int i** je promenljiva dostupna u get i set delu indeksera



# Indekseri

- ▶ Argument indeksera može biti proizvoljnog tipa (vrednosnog ili referentnog)

- ▶ class Recnik

```
{  
    private string[] srpski, engleski; // pretpostavimo da ova 2 niza čuvaju na  
                                     // pozicijama sa istim indeksom srpsku reč i njen prevod na engleski  
    public string this[string rec] {  
        get {  
            for (int i = 0; i < srpski.Length; i++) // tražimo reč u nizu srpskih reči  
                if (rec == srpski[i])              // ako je pronađena reč na srpskom  
                    return engleski[i];            // vraćamo odgovarajuću reč na engleskom sa pozicije i  
            return "";                               // ako reč nije pronađena vraćamo prazan string  
        }  
    }  
    ...  
}
```

# Indekseri

- ▶ Indeksers može imati i više od jednog argumenta

- ▶ class Matrica3D

```
{  
    private int[, ] matrica;  
    public int this[int i, int j, int k]  
    {  
        get { return matrica[i, j, k]; }  
        set { this.matrica[i, j, k] = value; }  
    }  
    ...  
}
```

# Indekseri

- ▶ Primer: projekat Poligoni

# Preklapanje operatora

- ▶ Unarni operatori koji se mogu preklopiti: `+`, `-`, `!`, `~`, `++`, `--`, `true`, `false`
- ▶ Binarni operatori koji se mogu preklopiti: `+`, `-`, `*`, `/`, `%`, `&`, `|`, `^`, `<<`, `>>`
- ▶ Operatori poređenja se mogu preklopiti: `==`, `!=`, `<`, `>`, `<=`, `>=`
  - ▶ Moraju se preklapati u parovima tj. ako se preklopi jedan operator iz para obavezno se preklapa i drugi: `==` i `!=`, `<` i `>`, `<=` i `>=`
- ▶ Uslovni logički operatori **ne mogu** se preklopiti: `&&`, `||`
  - ▶ Oni se dobijaju uslovnim izvršenjem operatora `&` ili `|` u kombinaciji sa `true` i `false` (nije greška - u C#u postoje operatori `true` i `false`) koji se mogu preklopiti
- ▶ Operator indeksiranja `[]` **ne može** se preklopiti
  - ▶ Moguće je definisati indeksere

# Preklapanje operatora

- ▶ Operator kastovanja **ne može** se preklopiti:  $(T)x$ 
  - ▶ Mogu se definisati operatori konverzije (eksplicitni i implicitni)
- ▶ Kombinovani operatori dodele **ne mogu** se preklopiti:  $+=, -=, *=, /=, \%=, \&=, |=, ^=, <<=, >>=$ 
  - ▶ Oni se evaluiraju kombinovanjem operatora  $=$  sa operatorima koji se mogu preklopiti:  $+, -, *, /, \%, \&, |, ^, <<, >>$
- ▶ Ovi operatori se **ne mogu** preklopiti:  $=, ., ?:, ??, ->, =>, f(x), \text{as}, \text{checked}, \text{unchecked}, \text{default}, \text{delegate}, \text{is}, \text{new}, \text{sizeof}, \text{typeof}$

# Preklapanje operatora

- ▶ Za razliku od C++a operatori se mogu zadati **isključivo kao statičke metode**
- ▶ Argumente operatorskih funkcija ne treba menjati u telu operatorske funkcije, operatorske funkcije obično vraćaju novi objekat deklarisanog tipa
- ▶ Operatorske funkcije nemaju podrazumevani argument **this**
- ▶ Ne postoje prijateljske funkcije klasa
- ▶ Pravila za preklapanje operatora data su sledećom tabelom gde je **operator** ključna reč, **#** neki od ranije pomenutih operatora za preklapanje, a **x** i **y** su operandi

Operatorski zapis	Funkcionalni zapis
# x	operator #(x)
x #	operator #(x)
x # y	operator #(x, y)

# Preklapanje operatora

## ► Primer kompleksni broj:

```
► public class KompleksniBroj
{
    public float re;
    public float im;
    public KompleksniBroj(float re, float im)
    {
        this.re = re;
        this.im = im;
    }
    ...
}
```

# Preklapanje operatora

Funkcija za definisanje binarnog operatora ima 2 argumenta

► ...

```
public static KompleksniBroj operator +(KompleksniBroj k1, KompleksniBroj k2)
{
    return new KompleksniBroj(k1.re + k2.re, k1.im + k2.im);
}
```

Ključna reč **operator** za zadavanje operatorskih funkcija

```
public static KompleksniBroj operator ++(KompleksniBroj k)
{
    return new KompleksniBroj(k.re + 1, k.im + 1);
}
...
```

Funkcija za definisanje unarnog operatora ima 1 argument. Ne zadaju se posebne funkcije za prefiksni i postfiksni poziv unarnog operatora, odgovarajuće ponašanje za jedan i drugi poziv obezbeđuje sam kompajler tako što istu funkciju poziva na različite načine prefiksno i postfiksno.



# Preklapanje operatora

- ▶ Primer: projekat KompleksniBrojeviOperatori
- ▶ Primer: projekat Matrice

# Parcijalne klase

- ▶ Moguće je implementaciju jedne klase podeliti u više .cs fajlova radi bolje preglednosti.
- ▶ U tom slučaju u svim fajlovima klasa se označava ključnom rečju **partial**
  - ▶ Sintaksa: **partial class** *MojaKlasa*
- ▶ Svi delovi klase moraju biti u okviru istog *namespace*-a.
- ▶ Članovi klase definisani u jednom delu klase su ravnopravno dostupni u svim ostalim delovima klase.
- ▶ Koristi se npr. kod programiranja grafičkih komponenti kada jedan fajl sadrži kod koji piše programer, a drugi fajl sadrži kod koji je generisan od strane Visual Studija i programer ne sme da menja taj kod (obrađićemo detaljnije kod Windows aplikacija).
- ▶ **Primer: projekat ParcijalneKlase**

# Klasa System.Object

- ▶ Osnovna klasa za sve ostale vrednosne i referentne tipove u jeziku C#
- ▶ Ima samo 1 konstruktor: `public Object()`
- ▶ Metode:
  - ▶ `public Type GetType()` - Vraća tip tekućeg objekta preko instance klase Type.
  - ▶ `public virtual string ToString()` - Konvertuje tekući objekat u njegovu stringovsku reprezentaciju (slično kao `toString()` u Javi).

Razlika u odnosu na Javu je to što je slovo "T" veliko i što je obavezna ključna reč "override".

Osnovna implementacija iz klase Object vraća naziv tipa kog je taj objekat. U velikoj većini klasa iz .NET Frameworka ova metoda je predefinisana tako da vraća neke konkretnije informacije o samom objektu.

# Klasa System.Object

## ► Metode:

- `public static bool ReferenceEquals(object objA, object objB)` - Vraća true ako `objA` i `objB` pokazuju na isti objekat u dinamičkoj zoni memorije tj. ako su `objA` i `objB` iste reference. U suprotnom vraća false.

- Implementacija iz klase Object:

```
public static bool ReferenceEquals (Object objA, Object objB)
{
    return objA == objB;
}
```

- **ReferenceEquals ne treba pozivati nad vrednosnim tipovima** jer zahteva argumente tipa Object. Čak i ako se prosledi npr. isti `int` kao argument biće upakovan (*boxing*) u dva različita objekta pa **ova metoda uvek vraća false za vrednosne tipove**.

# Klasa System.Object

- ▶ Kako radi operator `==` (nije deo klase Object, ali ga sve equals metode koriste pa je važno da razumemo kako on radi)
  - ▶ Za ugrađene vrednosne tipove iz .NET Frameworka operator jednakosti `==` je predefinisano tako da vraća `true` ako su `vrednosti` oba operanda jednake. U suprotnom vraća `false`.
  - ▶ Za ugrađene referentne tipove (osim nepromenljivih *immutable* tipova kao što je npr. `string`), `==` vraća `true` ako oba operanda pokazuju na isti objekat. U suprotnom vraća `false`. Za tip `string` operator `==` je predefinisano tako da radi isto kao za vrednosne tipove poređenje po vrednosti.
  - ▶ Za tipove koje sami pišemo (vrednosne ili referentne) možemo da predefinišemo operator `==` (detaljnije objašnjenje u oblasti o operatorima)

# Klasa System.Object

## ► Metode:

- `public virtual bool Equals(object obj)` - Vraća true ako je tekući objekat `this` jednak sa objektom `obj`, inače vraća false.

Ako se ne override-uje ova metoda

- za vrednosne tipove radi poređenje po vrednosti;
- za referentne tipove radi isto što i statička metoda `ReferenceEquals` tj. radi poređenje po referenci.

Često je potrebno da za naše klase radimo override ove metode.

- `public virtual int GetHashCode()` - Vraća heš ključ datog objekta za njegovo smeštanje u heš tablicu. Biće detaljno objašnjena uz klasu `Dictionary`.

## ► Primer: projekat KlasaObject

# Klasa System.Object

## ► Metode:

- `public static bool Equals(object objA, object objB)` - Statička je pa za razliku od prethodne dozvoljava da prvi argument bude null. Implementacija iz .NET Frameworka

```
public static bool Equals(Object objA, Object objB) {  
    if (objA == objB) { // radi prvo proveru operatorom == slično kao ReferenceEquals  
        return true; // ovo se izvršava i ako su oba null  
    }  
    if (objA==null || objB==null) { // ako je samo jedan objekat null vraća false  
        return false;  
    }  
    return objA.Equals(objB); // ako su oba različita od null zove se Equals  
}
```

# Klasa System.Object - online izvorni kod i dokumentacija

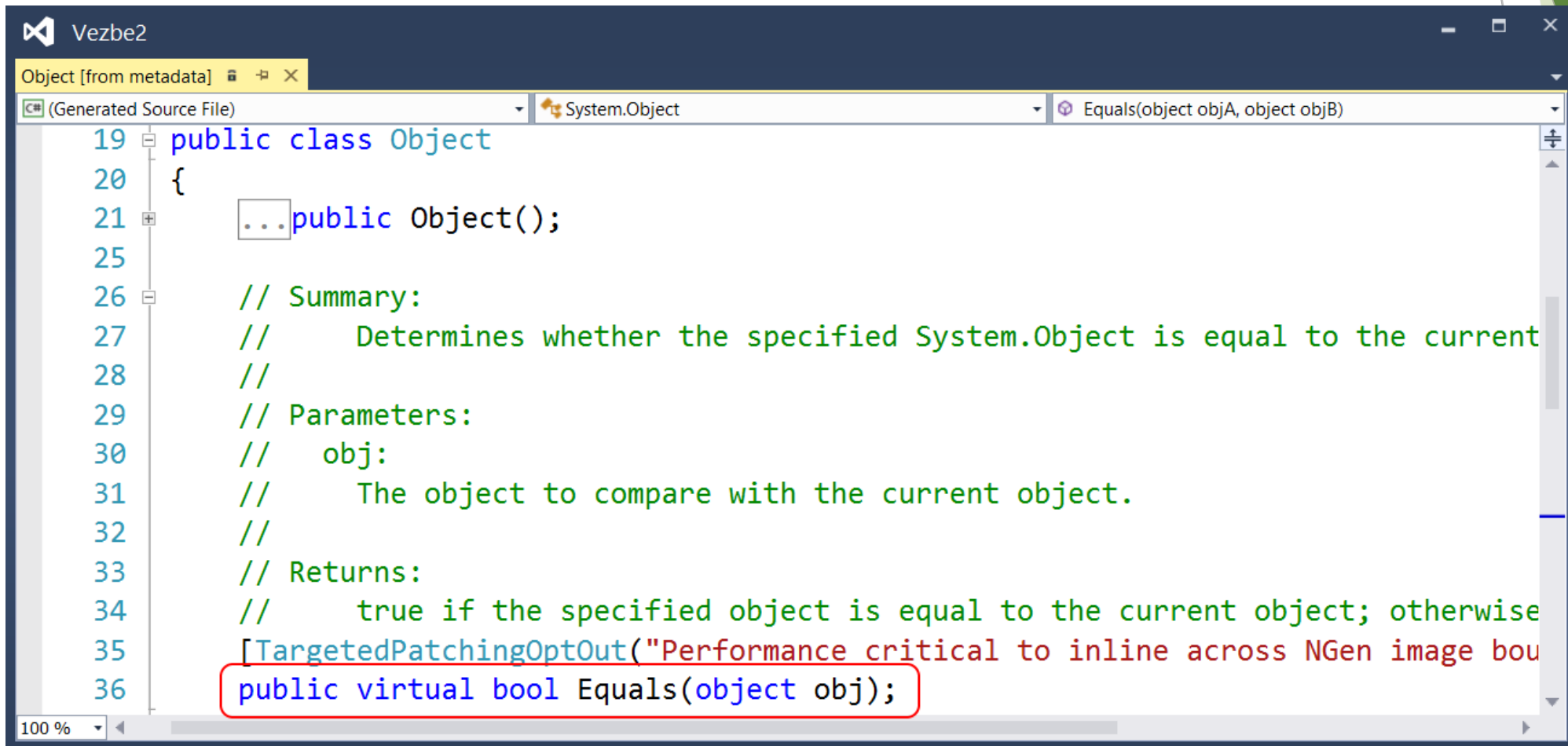
- ▶ Microsoft je omogućio slobodni pristup izvornom kodu .NET Framework-a pa npr. Implementaciju klase Object možemo da pronađemo na:  
<https://referencesource.microsoft.com/#mscorlib/system/object.cs>
- ▶ Na istoj stranici možemo da pronađemo i linkove ka implementaciji ostalih klasa iz .NET Frameworka.
- ▶ Zvanična dokumentacija za klasu Object dostupna je na (u okviru iste strane dostupna je pretraga pomoću koje može da se pronađe dokumentacija za ostale klase):  
<https://docs.microsoft.com/en-us/dotnet/api/system.object?view=netframework-4.8>



# Klasa System.Object - dokumentacija u okviru Visual Studija

- ▶ Bez pristupa internetu možemo u okviru Visual Studija da vidimo dokumentacione komentare u bilo kojoj ugrađenoj klasi iz .NET-a.
- ▶ Ako negde u našem kodu selektujemo deklaraciju promenljive tipa npr. Object ili poziv neke metode ovog tipa npr. Equals možemo desnim klikom pa izborom opcije “Go To Definition” (ili direktno prečicom F12) da vidimo dokumentacione komentare (ne i izvorni kod) iz te klase kao na sledećoj stranici.

# Klasa System.Object - dokumentacija u okviru Visual Studija



The screenshot shows the Visual Studio IDE with a window titled 'Object [from metadata]'. The 'Generated Source File' tab is active, displaying the source code for the `System.Object` class. The `Equals(object objA, object objB)` method is selected in the Solution Explorer. The code is as follows:

```
19 public class Object
20 {
21     ...public Object();
25
26     // Summary:
27     //     Determines whether the specified System.Object is equal to the current
28     //
29     // Parameters:
30     //     obj:
31     //     The object to compare with the current object.
32     //
33     // Returns:
34     //     true if the specified object is equal to the current object; otherwise
35     [TargetedPatchingOptOut("Performance critical to inline across NGen image bou
36     public virtual bool Equals(object obj);
```

The line `public virtual bool Equals(object obj);` is highlighted with a red rectangle. The status bar at the bottom indicates '100 %'.