Objektno programiranje u C# jeziku

Katedra za informatiku
Fakultet tehničkih nauka
Univerzitet u Novom Sadu

Objektno programiranje

- Identifikuje entitete sa kojima program operiše
- Za entitete se identifikuju
  - podaci koji se evidentiraju
  - operacije koje se mogu izvršavati nad entitetom

Objektno programiranje

- Apstrakcija
  - Zanemaruju se osobine entiteta koje nisu važne za konkretan problem
- Klasifikacija
  - Entiteti se grupišu po tipu
  - Time su identifikovane klase entiteta
  - Pojedinačni entiteti su primerci svoje klase

Objekat vs Klasa

- Objekat je konkretan entitet
  - Ima podatke koji opisuju njegove osobine
  - Nad tim podacima se može izvršiti programski kod
- Klasa je šablon koji opisuje sve entitete određenog tipa i definiše
  - Koje osobine svi entiteti tog tipa imaju
    - To su atributi klase
  - Programski kod koji se može izvršiti nad entitetima tog tipa
    - To su metode klase

Objekat vs Klasa

- Program u izvršavanju radi sa objektima
  - Objekat ima svoj životni ciklus
  - U toku životnog ciklusa
    - skladišti i menja podatke
    - izvršava se programski kod nad podacima
- Klasa je apstrakcija da predstavimo koje podatke skladištimo i koje operacije izvršavamo nad objektima određenog tipa
  - Objekat je primerak (instanca) klase

C# Klase

- Deklaracija klase

```
class Krug {
  //sadržaj klase
}
```

- Ime klase može biti različito od imena fajla
- Jedan fajl može sadržati više klasa
- Instanciranje klase
  - `Krug k = new Krug();`

C# Članovi klase – atributi i metode

- Atributi
- Metode
- Svojstva
- Ugrađene klase

C# Atributi klase

- `private double poluprecnik;`
- Atribut se može inicijalizovati pri deklaraciji
  - `private Tacka centar = new Tacka(0,0);`
  - Kompajler automatski dodaje kod za inicijalizaciju na početak konstruktora

C# Metode klase

```
public String ispisiIme(bool malaSlova)
{
   if (malaSlova)
         return ime.ToLower();
   else
         return ime;
}
```

Preklapanje metoda (overloading)

- Može da postoji više metoda istog imena, ali sa različitim argumentima

```
public double ispisiIme(bool malaSlova)
{
   ...
}
public double ispisiIme()
{
   ...
}
```

- Ne može da bude više metoda sa istim imenom i argumentima, ali različitog tipa povratne vrednosti

C# Svojstva (Properties) [1]

- Enkapsulacija
  - program koji koristi klasu ne treba da poznaje detalje implementacije te klase
  - atributi ne trebaju direktno da budu dostupni spolja
- Standardan mehanizam enkapsulacije – get/set metode
- Properties
  - obezbeđuju efikasniji mehanizam enkapsulacije
  - get/set metode koje se koriste kao atributi

```
private double x;
public double X
{
    get { return x; }
    set { x = value; }
}
```

  - promenljiva `value` sadrži vrednost koja je prosleđena i koja se postavlja u atribut
  - Atribut može biti *read-only* ili *write-only* ako se implementira samo get ili set deo koda

C# Svojstva (Properties) [2]

- Property je metoda sa specifičnom sintaksom, što znači da može da sadrži proizvoljan kod
- Automatski property – jednostavna sintaksa za slučaj kada se samo postavlja, odnosno preuzima vrednost atributa
  - `public double X { get ; set; }`
- pristup property elementu

```
Tacka t = new Tacka();
t.X = 5.2;
```

Prostori imena (namespaces)

- Tipovi su organizovani u prostore imena zbog
  - jedinstvene identifikacije tipova
  - organizacije tipova po srodnosti
- Slično kao paketi u Javi, ali .NET prostori imena predstavljaju logičku strukturu koja ne mora biti u skladu sa strukturom fajlova i foldera
- Jedinstvena identifikacija tipa
  - Prostor imena + naziv tipa

```
System.Collections.ArrayList l = new
  System.Collections.ArrayList();
```

- Ovakvo korišćenje punog imena bi smanjilo čitljivost koda, zato se koristi ključna reč `using`

```
using System.Collections;
ArrayList l = new ArrayList();
```

Nasleđivanje

- Različite klase mogu da dele deo osobina i operacija
- Zbog iskorišćenja koda zajednički atributi i metode se mogu grupisati
- Sve klase u C# implicitno nasleđuju klasu **`System.Object`** sa metodama
  - **`ToString()`** – konvertuje sadržaj objekta u tekst
  - **`Equals(Object )`** – poredi sadržaj objekta sa drugim objektom
  - **`GetHashCode()`** – vraća *hash code* objekta (sadržaj objekta mapiran na numeričku vrednost)
  - **`GetType()`** – Vraća **`Type`** objekat koji opisuje klasu kojoj objekat pripada

Nasleđivanje

- Sintaksa – koristi se karakter :

```
public class Osoba
{
  protected String ime;
  protected String prezime;
}
public class Radnik : Osoba
{
  protected String radnoMesto;
  public void obracunajPlatu() {…}
}
```

Polimorfizam

- Objekat se može posmatrati kao da je višestrukog tipa (polimorfno)
- Implicitna konverzija – iz nasleđenog tipa u bazni tip

```
Student s1 = new Student();
s1.Prosek = 9.3;
Object obj = s1;
```

  - U ovom slučaju promenljiva tipa bazna klasa može da pristupi (bez eksplicitne konverzije) samo članovima definisanim u baznoj klasi
    - `obj.ToString()` – ispravno
    - `obj.Prosek` – greška pri kompajliranju
- Eksplicitna konverzija – iz baznog tipa u nasleđeni tip
  - `Student s2 = (Student) obj;`
  - Moguć izuzetak u toku izvršavanja (`InvalidCastException`), ako instanca nije odgovarajućeg tipa
  - `Racunar r = (Racunar) obj;` - **IZUZETAK!** (U toku kompajliranja nije poznat tip instance na koju pokazuje promenljiva `obj`. U toku izvršavanja desiće se izuzetak, jer promenljiva tipa `Racunar` ne može pokazivati na instancu tipa `Student` (na koju pokazuje `obj`).

Nasleđivanje i konverzija tipova

- Pojava run-time izuzetka pri konverziji, može se sprečiti operatorima:
  - **`is`** - utvrđuje tip instance. Najpre provera tipa, pa zatim konverzija

```
if (obj is Racunar)
        Racunar r = (Racunar) obj;
```

  - **`as`** – istovremeno provera i konverzija. Ukoliko konverzija nije moguća, operator vraća vrednost **`null`**

```
Racunar r = obj as Racunar;
```

Dynamic binding

- Ključna prednost objektnog programiranja
- Bez dynamic binding-a objektno programiranje bi bilo samo način organizacije koda
  - Sama modularizacija koda je i u proceduralnom programiranju moguća kroz odvajanje srodnih promenljivih i funkcija u odvojene celine/fajlove
- Omogućuje nezavisno i naknadno proširivanje funkcionalnosti bez izmene postojećeg koda

Dynamic binding

- Ako postoji jedna osnovna klasa koju nasleđuje više klasa naslednica
- Dynamic binding predstavlja sposobnost programskog jezika da tretira objekte naslednica zavisno od njihovog tipa
- Programski kod koji radi sa objektima je generički, radi sa osnovnom klasom i ne sadrži kod specifičan za klase naslednice
- U toku izvršavanja programa (*run-time*), poziva se odgovarajuća metoda klase naslednice zavisno od tipa objekta
  - dynamic (late, run-time) binding

Dynamic binding

```
List<Osnovna> l;
l.add(new Naslednica1());
l.add(new Naslednica2());
```

```
foreach (Osnovna x in l)
{
    x.izvrsi();
}
```

Ovaj deo koda uvek ostaje isti

Poziva se metoda izvrsi iz klase Naslednica1 ili Naslednica2 zavisno od tipa objekta x

Dynamic binding

- Na taj način kod koji radi sa osnovnom klasom ne treba da se menja
- Samo se u klasama naslednicama definiše novo ponašanje

Virtuelne metode i redefinisanje metoda (overriding)

- Metoda pretka čije ponašanje može biti u klasi naslednici izmenjeno (redefinisano) metodom koja ima istu deklaraciju
- To je preduslov da bi *dynamic binding* funkcionisao
  - Klasa predak treba da ima definisanu metodu
  - Klasa naslednica ima istu tu metodu sa drugačijim ponašanjem
  - U klasi predak metoda mora biti definisana kao virtuelna

Virtuelne metode i redefinisanje metoda (overriding)

- Virtualna metoda se deklariše sa **`virtual`**
  - Za razliku od Jave gde su sve metode implicitno virtualne
- Redefinisana metoda se deklariše sa **`override`**
  - Ako se ne stavi override, metoda se ne smatra redefinicijom metode pretka nego kao nova metoda sa istom deklaracijom
  - Kompajler prikazuje upozorenje u ovom slučaju
  - Rečju **`new`** može se reći kompajleru da je ovo svesno urađeno
  - Virtuelna metoda ne može biti **`private`**
  - Metode i redefinicije moraju imati potpuno istu deklaraciju i modifikator pristupa

Apstraktne klase i metode

- Ne može biti instancirana
- Služe kao osnov za nasleđivanje
- Sintaksa

```
public abstract class Figura {
…
}
```

- Može da sadrži apstraktne metode
  - metoda deklarisana bez implementacije
  - klase naslednice zadužene da implementiraju telo metode
  - sintaksa

```
public abstract double getPovrsina();
```

Interfejsi

- specifikacija funkcionalnosti koje određeni objekat podržava, bez implementacije ovih funkcionalnosti
- implementaciju funkcionalnosti definisana je u klasi koja implementira interfejs
- Klasa može da implementira više interfejsa (ali može da nasledi samo jednu klasu)
- u sintaksnom smislu interfejs je apstraktna klasa koja sadrži samo javne apstraktne metode
- Deklariše se rečju **`interface`**
- Ispred metoda nema modifikatora pristupa, jer su uvek javne

Interfejsi - sintaksa

```
interface IPoredjenje
{
   int poredi(Object obj);
}
…
class Student: IPoredjenje
{
   int poredi(Object obj)
   {
      …
   }
}
```

Interfejsi

- Interfejs ne sme imati atribute, čak ni statičke
- Nema konstruktora
- Nema ugrađenih tipova i klasa unutar interfejsa
- Ne može da nasledi klasu, ali može da implementira drugi interfejs

Statičke klase i članovi klase

- Statički članovi klase
  - funkcionalnosti koje nisu vezane za neku konkretnu instancu klase
  - može im se pristupati bez kreiranja objekta klase

```
class Math {
        ...
    public static double Sqrt(double d) { … }
}
...
double v = Math.Sqrt(2);
```

    - u statičkoj metodi moguće je pristupati samo statičkim atributima i pozivati samo statičke metode
    - konstanta je statički član koji se ne može menjati

```
public const double PI =
3.14159265358979323846
```

- Statička klasa
  - sadrži samo statičke atribute i metode
  - ne može da se instancira

```
public static class Convert
```

Modifikatori pristupa

- Za klasu
  - `internal` – podrazumevani modifikator, pristup klasi moguć iz drugih klasa unutar istog sklopa (eng. *assembly*)
  - `public` – pristup klasi moguć iz svih drugih klasa
  - ugrađene klase mogu imati modifikatore pristupa kao atributi
- Za atribut
  - `private` – podrazumevani modifikator, atribut dostupan samo unutar klase
  - `protected` – atribut dostupan unutar klase i iz klasa naslednica
  - `internal` – atribut dostupan iz klasa unutar istog sklopa (eng. *assembly*)
  - `protected internal` – atribut dostupan iz klasa unutar istog sklopa i iz klasa naslednica
  - `public` – atribut dostupan iz svih klasa

Zapečaćene (sealed) klase i metode

- Zapečaćena klasa je klasa koju nije moguće naslediti
- Klasa eksplicitno zabranjuje nasleđivanje rečju **`sealed`** u deklaraciji

```
sealed class A
{
   …
```

- Zapečaćena metoda je redefinisana metoda koju nije dalje moguće redefinisati

```
public sealed override int m1() { … }
```

- **`sealed`** se odnosi samo na redefinisane metode, jer za obične metode izostavljanjem reči **`virtual`** moguće je zabraniti njenu redefiniciju

Ugrađene klase

- Klasa definisana unutar neke klase
- Koristi se kada se klasa koristi samo za realizaciju funkcionalnosti klase unutar koje je definisana
- Najčešće spolja nije vidljiva i nije namenjena za korišćenje iz drugih delova koda
  - Zato je default modifikator private
  - Mogu se staviti i drugi modifikatori

Ugrađene klase

```
class Glavna
{
  class Unutrasnja
 {
   ...
 }
   ...
}
```

Anonimne klase

- Moguće je definisati novu klasu u trenutku kreiranja objekta te klase
  - Ova klasa nema ime
  - Pri kreiranju objekta navode se svojstva klase i vrednosti svojstava u objektu

```
var a = new {Cena = 450.37, Naziv = "TV"};
a.Cena = 500.99;
```

Operatori nad objektima

- Naziv tipa
  - `typeof`
- Tip objekta
  - `is`
- Konverzija tipa (cast)
  - `as`

- Može se vršiti preklapanje operatora

Metode koje proširuju klasu Extension Methods

- Moguće je naknadno proširiti klasu metodama bez izmene izvornog koda klase
- Koristi se za klase preuzete iz postojećih biblioteka
- Time nad objektima klase možemo pozivati naknadno dodate metode
  - Bez Extension Methods, jedina varijanta za uvođenje novih metoda je nasleđivanje postojeće klase, ali se onda svugde mora koristiti klasa naslednica

Metode koje proširuju klasu Extension Methods

- Sintaksa
  - Npr. proširujemo klasu String novom metodom koja izbacuje sva prazna mesta iz sredine stringa
  - Uvodimo novu klasu u kojoj ćemo definisati metodu

```
static class StringProsirenje
{
  public static string TrimMiddle(this string s)
  {
     return s.Replace(" ", "");
  }
}
```

Metode koje proširuju klasu Extension Methods

```
string s = "Marko Markovic Novi Sad";
Console.WriteLine("Spojeno sve: " +
  s.TrimMiddle());
```

- Nakon definisanja metode koja proširuje klasu, kompajler tretira ovu metodu isto kao i druge metode koje su definisane u samoj klasi