

# Java tečaj

2. dio

Razredi i objekti

# Java razredi

Razred (engl. *class*) predstavlja poopćenje "C-struktura"

- ◆ Na elementarnoj razini, razred je struktura, koja osim članskih varijabli ima i metode za inicijalizaciju (konstruktori), vlastite funkcije ("metode") te kontrolu pristupa (tko može pristupiti čemu)

# Java razredi

Primjerak razreda (objekt, "*istanca*")

- ◆ Dio memorije alociran za pohranu nestatičkih članskih varijabli razreda (+ još nešto sitno)
- ◆ Objekti se stvaraju na gomili (engl. heap); ne možemo ih stvarati na stogu
- ◆ Radimo s referencama na objekte

# Java razredi

## Razred

- ◆ Statičke članske varijable razreda čuvaju se na jednom zajedničkom mjestu – dijele ih svi primjerci razreda
- ◆ Programski kod metoda također se čuva na jednom mjestu – dijele ga svi primjerci razreda

# Java razredi

- ◆ Definirane su posebne metode:
  - Za stvaranje primjerka razreda:  
*konstruktori*
  - Primjerke ne možemo uništavati; skupljač smeća će ih automatski osloboditi kad naš kod izgubi reference na njih
  - Prije no što objekt bude uništen, poziva se metoda `finalize()`: ne koristiti!

# Java razredi

## ◆ Primjer:

```
public class GeometrijskiLik {  
    /** Privatni element koji pohranjuje ime lika */  
    private String ime;  
    /** Konstruktor geometrijskog lika */  
    public GeometrijskiLik(String ime) {  
        this.ime = ime;  
    }  
    /** Dohvat imena geometrijskog lika */  
    public String getIme() {  
        return this.ime;  
    }  
}
```

# Java razredi

```
/** Dohvat opsega geometrijskog lika */  
public double getOpseg() {  
    return 0;  
}  
/** Dohvat površine geometrijskog lika */  
public double getPovrsina() {  
    return 0.0;  
}  
}
```

# Java razredi

## ◆ Primjer uporabe:

```
public class Primjer1 {  
  
    public static void main(String[] args) {  
  
        GeometrijskiLik lik1 = new GeometrijskiLik("Lik1");  
        GeometrijskiLik lik2 = new GeometrijskiLik("Lik2");  
  
        System.out.println("Ime prvog lika je "+lik1.getIme());  
        System.out.println("Ime drugog lika je "+lik2.getIme());  
  
    }  
}
```



# Java razredi

- ◆ Primjer uporabe:

```
GeometrijskiLik lik1 = new GeometrijskiLik("Lik1");
```

- ◆ Varijabla `lik1` je po vrsti referenca (slično kao pokazivač u C-u)
- ◆ Operator **new** alocira u memoriji mjesto za jedan primjerak razreda i zatim zove odgovarajući konstruktor koji će inicijalizirati objekt; vraća referencu na novi objekt

# Java razredi

## ◆ Primjer uporabe:

```
GeometrijskiLik lik1 = new GeometrijskiLik("Lik1");  
GeometrijskiLik lik2 = lik1;
```

- ◆ "lik1" i "lik2" su dvije reference koje pokazuju na isti objekt u memoriji!

# Java razredi

- ◆ Pravokutnik je poseban geometrijski lik:

```
public class Pravokutnik extends GeometrijskiLik {  
    /** X koordinata gornjeg lijevog vrha. */  
    private int vrhX;  
    /** Y koordinata gornjeg lijevog vrha. */  
    private int vrhY;  
    /** Sirina pravokutnika. */  
    private int sirina;  
    /** Visina pravokutnika. */  
    private int visina;
```

# Java razredi

- ◆ Pravokutnik je poseban geometrijski lik:

```
/**
 * Konstruktor pravokutnika
 */
public Pravokutnik(int vrhX, int vrhY, int sirina, int visina)
{
    super("Pravokutnik"); // Poziv konstruktora od g. lika
    this.vrhX = vrhX;
    this.vrhY = vrhY;
    this.sirina = sirina;
    this.visina = visina;
}
```

# Java razredi

- ◆ Pravokutnik je poseban geometrijski lik:

```
/**  
 * Dohvat X-koordinate gornjeg lijevog vrha  
 */  
public int getVrhX() {  
    return vrhX;  
}  
// ostale metode...
```

# Java razredi

- ◆ Pravokutnik je poseban geometrijski lik:

```
/**  
 * Izračun opsega pravokutnika; ova metoda prekriva  
 * istu metodu definiranu u razredu GeometrijskiLik  
 */  
public double getOpseg() {  
    return (double)(2*sirina + 2*visina);  
}  
// ostale metode...
```

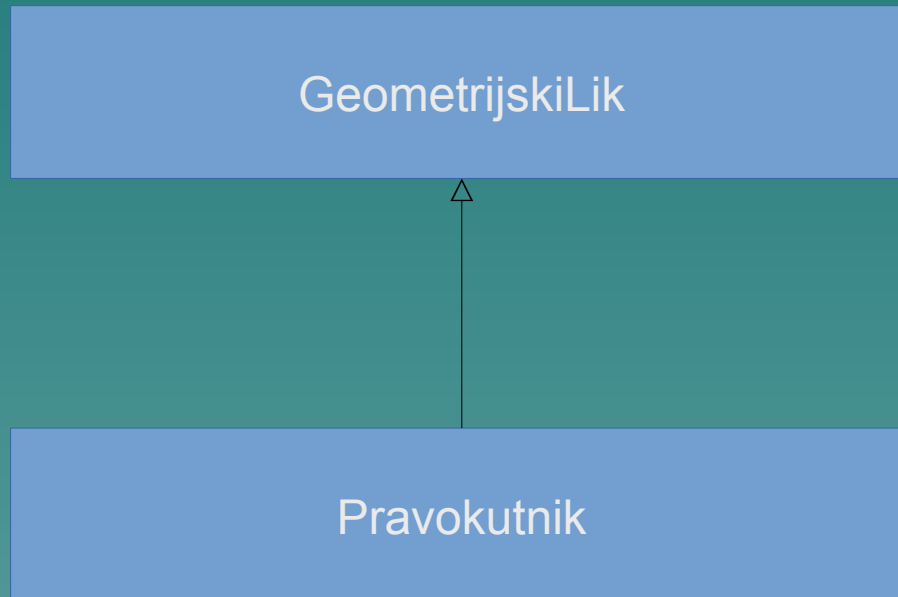
# Java razredi

- ◆ Pravokutnik je poseban geometrijski lik:

```
/**  
 * Izračun površine pravokutnika; ova metoda prekriva  
 * istu metodu definiranu u razredu GeometrijskiLik  
 */  
public double getPovrsina() {  
    return sirina*visina;  
}  
}
```

# Java razredi

- ◆ Dijagram razreda:
  - Uočite “praznu” strelicu





# Java razredi

- ◆ Uočite kako Pravokutnik i GeometrijskiLik imaju svaki svoju definiciju metode `getPovrsina()`.
- ◆ Mogućnost da razred Y koji nasljeđuje razred X redefinira neku metodu razreda X (engl. *override*) naziva se **polimorfizam**.

# Java razredi

- ◆ Pojam **polimorfizam** također označava mogućnost jezika da dopusti definiranje više funkcija koje se isto zovu, ali imaju različite argumente.
- ◆ Tada će se prilikom poziva određene metode utvrditi koju točno inačicu metode treba pozvati.

# Java razredi

Pogledajte sljedeći kod:

```
Pravokutnik p1 = new Pravokutnik(  
    "Lik1", 1, 1, 5, 5);
```

OK

```
Pravokutnik p2 = p1;
```

nepotrebno

```
GeometrijskiLik l3 = (GeometrijskiLik)p1;
```

upcast

```
GeometrijskiLik l4 = p1;
```

```
Pravokutnik p3 = (Pravokutnik)l3;
```

downcast

```
Pravokutnik p4 = l3;
```

Greška pri  
prevođenju

Nužno, moguća  
greška pri izvođenju  
ovisno na što l3  
pokazuje

# Java razredi

**Upcast** (ukalupljivanje prema roditeljskim razredima) je uvijek moguće; ne zahtijeva eksplicitno ukalupljivanje

**Downcast** (ukalupljivanje prema djeci) zahtijeva eksplicitno ukalupljivanje da bi se prevelo; može rezultirati iznimkom pri izvođenju

# Java razredi

Pri prevođenju koda, kada nad nekom referencom pozovemo metodu, prevodilac "vidi" samo metode vidljive kroz tip reference

Kako se generira polimorfni poziv, bit će pozvana metoda koja odgovara stvarnom razredu objekta: nije bitno kroz koji ga tip gledamo

# Java razredi

Pogledajte sljedeći kod:

```
Pravokutnik p1 = new Pravokutnik(  
    "Lik1", 1, 1, 5, 5);
```

```
GeometrijskiLik l3 = p1;
```

```
System.out.println(p1.getOpseg());
```

```
System.out.println(l3.getOpseg());
```

Oba će poziva pozvati `Pravokutnik::getOpseg` jer je vrsta objekta određena pri njegovoj konstrukciji (`new Pravokutnik(...)`) pa nije bitno kako gledamo na taj objekt.

# Java razred Object

- ◆ Java definira razred `Object` koji ima niz metoda
- ◆ Nama interesantne su:
  - `Object()`; - konstruktor bez argumenata
  - `int hashCode()`; - računa hash vrijednost objekta
  - `boolean equals(Object o)`; - usporedba s drugim objektom
  - `String toString()`; - vraća tekstualni opis objekta
- ◆ Svaki razred u Javi implicitno nasljeđuje razred `Object`

# Java razredi

## ◆ Dopunimo razred GeometrijskiLik:

```
public boolean equals(Object obj) {  
    if( !(obj instanceof GeometrijskiLik) ) return false;  
    GeometrijskiLik drugi = (GeometrijskiLik)obj;  
    return ime.equals(drugi.ime);  
}  
  
public String toString() {  
    return "Lik "+ime;  
}  
  
public int hashCode() { return ime.hashCode(); }
```



# Java razredi

## ◆ Dopunimo razred Pravokutnik:

```
public boolean equals(Object obj) {  
    if( !(obj instanceof Pravokutnik) ) return false;  
    Pravokutnik drugi = (Pravokutnik)obj;  
    return vrhX==drugi.vrhX && vrhY==drugi.vrhY &&  
        sirina==drugi.sirina && visina==drugi.visina;  
}  
public String toString() {  
    return super.toString() + "("+vrhX+","+vrhY+","+  
        sirina+","+visina+")";  
}  
public int hashCode() {  
    return Objects.hash(getIme(),vrhX,vrhY,sirina,visina);  
}
```

# Java razredi

## ◆ Primjer uporabe:

[illegible]

# Java razredi

## ◆ Primjer uporabe:

```
public class Primjer3 {  
  
    public static void main(String[] args) {  
  
        String s1 = new String("Ovo je tekst.");  
        String s2 = new String("Ovo je tekst.");  
        System.out.println("s1==s2 "+(s1==s2));  
        System.out.println("s1.equals(s2) "+s1.equals(s2));  
    }  
}
```

# Java razredi

- ◆ Nasljeđuje li Kružnica Elipsu?
- ◆ To je važno pitanje za OO dizajn!
- ◆ LSP: Liskov Substitution Principle:
  - Osnovne tipove mora se moći zamijeniti izvedenim tipovima
- ◆ Pročitati u knjizi podpoglavlje “Liskovino načelo supstitucije” (počinje na stranici 210).

# Java razredi

- ◆ Modifikatori zaštite privatnosti:
  - **private**: vidljivo samo kodu u istom razredu
  - bez (zovemo package-private): dodatno vide i razredi istog paketa
  - **protected**: dodatno vide i razredi koji nasljeđuju trenutni razred
  - **public**: dodatno vide i svi ostali


# Upravljanje pogreškama

- ◆ Što napraviti kada se u funkciji dogodi greška?
  - Prekinuti izvođenje programa
    - ◆ Loše! Zamislamo da je greška nastupila u nekoj biblioteci koju koristi naš program. Ista će srušiti program iako je programer možda htio obavijestiti korisnika da operacija nije uspjela i dalje nastaviti s radom.

# Upravljanje pogreškama

- ◆ Što napraviti kada se u funkciji dogodi greška?
  - Vratiti status pogreške
    - ◆ Loše! Pozivatelj mora provjeravati status. Ako on ne zna kako napraviti obradu, on svom pozivatelju treba vratiti nekakav status koji dalje treba porvjeravati...
    - ◆ Kako vratiti istovremeno rezultat i status?

# Upravljanje pogreškama

- ◆ Modernije rješenje: koncept iznimke (engl. *exception*)
  - ◆ Ako metoda regularno završi, sigurno vraća podatak
  - ◆ Ako se dogodi pogreška, izaziva se iznimka i započinje postupak obrade iznimke
  - ◆ Iznimku netko mora uhvatiti
- 



# Upravljanje pogreškama

- ◆ Neuhvaćene iznimke rezultiraju prekidom izvođenja programa
- ◆ Obradu radimo blokom `try-catch-finally`

```
String unos = null;
```

```
try {  
    unos = reader.readLine();  
} catch (IOException e) {  
    e.printStackTrace();  
    System.exit(1);  
}
```

- ◆ Metoda `readLine` izaziva `IOException`

# Upravljanje pogreškama

- ♦ Svaka metoda koja može izazvati iznimku, mora:
  - Tu iznimku obraditi (`try-catch` blok), ili
  - Deklarirati da izaziva tu iznimku

```
public int procitaj() throws IOException
{
    ...
}
```

- ♦ Izuzetak od pravila su unchecked iznimke (npr. `NumberFormatException`)

# Upravljanje pogreškama

- ◆ Svaka metoda može po potrebi i izazvati neku iznimku, npr.:

- ```
public int procitaj() throws IOException  
{  
    // funkcija koja nešto čita  
    // ako ne može pročitati znak, izazovi pogrešku:  
    throw new IOException("Ne mogu pročitati znak!");  
}
```

# Upravljanje pogreškama

- ◆ Kako točno ide obrada iznimaka?
  - Pretpostavimo da imamo program u kojem je metoda **main** pozvala metodu **m1** koja je pozvala metodu **m2** koja je pozvala metodu **m3**
  - Neka se u metodi **m3** dogodi iznimka **E**

# Upravljanje pogreškama

- ◆ Kako točno ide obrada iznimaka?
  1. Najprije se provjerava obrađuje li tko iznimku **E** u metodi **m3**
  2. Ako metoda **m3** ne obrađuje iznimku **E**, metoda se napušta, i provjerava se obrađuje li metoda **m2** tu iznimku
  3. Ako metoda **m2** ne obrađuje iznimku **E**, metoda se napušta, i provjerava se obrađuje li metoda **m1** tu iznimku

# Upravljanje pogreškama

- ◆ Kako točno ide obrada iznimaka?
  4. Ako metoda **m1** ne obrađuje iznimku **E**, metoda se napušta, i provjerava se obrađuje li metoda **main** tu iznimku
  5. Ako metoda **main** ne obrađuje iznimku **E**, metoda se napušta, i program se terminira uz ispis poruke pogreške
  6. Ako bilo koja metoda na ovom putu uhvati tu pogrešku, program se nastavlja izvoditi od tog **catch** bloka

# Upravljanje pogreškama

## ◆ Struktura izraza za obradu pogreške

```
try {  
    ...  
} catch (SomeException1 e1) {  
    ...  
} catch (SomeException2 e2) {  
    ...  
} catch (SomeException3 e3) {  
    ...  
} finally {  
    ...  
}
```

# Upravljanje pogreškama

- ◆ Struktura izraza za obradu pogreške
  - Pri tome se blokovi **catch** pregledavaju od prvog prema zadnjem, i traži onaj koji obuhvaća izazvanu iznimku
  - Prvi koji je pronađen bit će izvršen; svi ostali se zanemaruju
  - Obratiti pažnju na stablo nasljeđivanja iznimaka



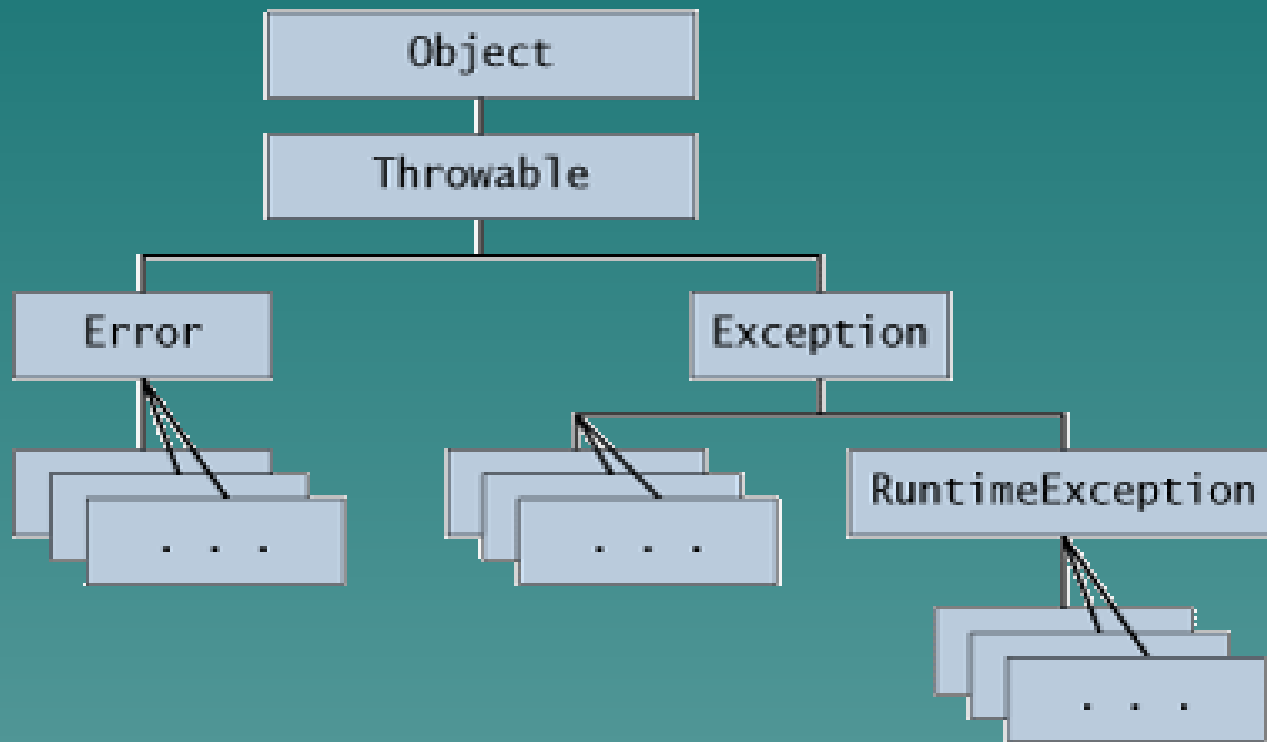
# Upravljanje pogreškama

## ◆ Besmisleni kod: krivi poredak!


```
try {  
    ...  
} catch (IOException e1) {  
    ...  
} catch (FileNotFoundException e2) {  
    ...  
} finally {  
    ...  
}
```

# Upravljanje pogreškama

## ◆ Stablo nasljeđivanja iznimaka



# Upravljanje pogreškama

- ◆ Možemo izvoditi vlastite iznimke: definiramo novi razred koji nasljeđuje neku od iznimaka
  - ◆ Sve što nasljeđuje RuntimeException modelira neprovjeravane iznimke
  - ◆ Sve što nasljeđuje Exception ali ne RuntimeException su provjeravane iznimke
- 

# Upravljanje pogreškama

- ♦ Struktura izraza za obradu pogreške
  - blok **finally** izvršava se uvijek po završetku izvođenja bloka **try**, nevezano uz način završetka (da li regularno, ili putem iznimke)
  - Idealno mjesto za kod koji oslobađa zauzete resurse (primjerice, zatvara otvorene datoteke i sl.)
- ♦ Proširenje je **try-with-resources**

# Provjera argumenata metode

- ◆ Ako metoda dobiva argument neprikladne vrijednosti, baciti `IllegalArgumentException` (ili neku specifičniju: `IndexOutOfBoundsException`)
- ◆ Ako je argument null i to nije dozvoljeno: diskutabilno
  - `IllegalArgumentException`,  
`NullPointerException`

# Provjera argumenata metode

- ◆ Konsenzus: bacati `NullPointerException`
- ◆ Štoviše, koristiti:

```
void metoda(String ime) {  
    this.ime = Objects.requireNonNull(ime);  
}  
  
void metoda(String ime) {  
    this.ime = Objects.requireNonNull(  
        ime, "ime ne smije biti null");  
}
```

# Upravljanje pogreškama

◆ Pročitati:

<https://docs.oracle.com/javase/tutorial/essential/exceptions/>