

# Java tečaj

2. dio

Razredi i objekti

# Strukture u C-u

- ◆ Za izradu kompleksnih tipova podataka, C nudi strukture
- ◆ 

```
typedef struct pravokutnik_str {  
    int poz_x;  
    int poz_y;  
    int sirina;  
    int visina;  
    char *ime;  
} pravokutnik;
```

# Strukture u C-u

- ◆ Svojstva ovog pristupa:
  - Svatko može mijenjati vrijednosti članova strukture
  - Nema nikakve kontrole postavljaju li se vrijednosti članova strukture na legalne vrijednosti

# Strukture u C-u

- ◆ Jedno moguće rješenje
  - Zabraniti direktno korištenje članova strukture
  - Napisati poseban skup funkcija koje treba koristiti

# Strukture u C-u

## ◆ Npr. Stvaranje novog pravokutnika

```
pravokutnik *pravokutnik_novi(int x, int y, int s, int v,  
                                char *ime) {  
    pravokutnik *novi = (pravokutnik*)malloc(  
                                                sizeof(pravokutnik));  
  
    if(novi == NULL) return NULL;  
    novi->ime = (char*)malloc(strlen(ime)+1);  
    strcpy(novi->ime, ime);  
    novi->poz_x = x;  
    novi->poz_y = y;  
    novi->sirina = s;  
    novi->visina = v;  
    return novi;  
}
```

# Strukture u C-u

## ◆ Npr. Uništavanje pravokutnika

```
void pravokutnik_unisti(pravokutnik *p) {  
    if(p == NULL) return;  
    free(p->ime);  
    free(p);  
    return;  
}
```

# Strukture u C-u

- ◆ Npr. Postavljanje/dohvat širine uz kontrolu predanog argumenta

```
void pravokutnik_postavi_sirinu(pravokutnik *p, int s) {  
    if(s < 1) return;  
    p->sirina = s;  
    return;  
}
```

```
int pravokutnik_dohvati_sirinu(pravokutnik *p) {  
    return p->sirina;  
}
```

# Strukture u C-u

- ◆ Ukoliko se radi samo s napisanim metodama:
  - Smanjuje se mogućnost pogreške u kodu (npr. funkcija za stvaranje napisana je samo na jednom mjestu)
  - Članovi neće biti postavljeni na nekonzistentne vrijednosti
  - Problem: kako osigurati da svatko koristi te metode?



# Strukture u C-u

- ◆ Karakteristike pristupa:
  - Postoji metoda za stvaranje pravokutnika, koja prima argumente potrebne za proces stvaranja
  - Postoji metoda za uništavanje pravokutnika, koja prima pokazivač na pravokutnik

# Strukture u C-u

- ◆ Karakteristike pristupa:
  - Postoje metode za rad s pravokutnikom, koje primaju pokazivač na pravokutnik, plus dodatne argumente (npr. željena širina)

# Strukture u C-u

## ◆ Još jedan problem:

- Kako postići specijalizaciju strukture, ili dodavanje novih polja strukturi?
- Npr. Kvadrat je specijalni pravokutnik – visina i širina su mu isti
- Kako izreći: “ja imam sve elemente kao i struktura X, i dodatno imam još elemente ...)

# Java razredi

- ◆ Java uvodi poopćene “C-strukture”: razred (engl. class)
- ◆ Na elementarnoj razini, razred je struktura, koja osim članskih varijabli ima i vlastite funkcije (“metode”), te nudi kontrolu pristupa (tko može pristupiti čemu)

# Java razredi

- ◆ Definirane su posebne metode:
  - Za stvaranje primjerka razreda služi konstruktor (ekvivalent funkcije pravokutnik\_novi)
  - Za uništavanje primjerka, tj. objekta u Javi ne postoji posebna metoda – nema potrebe
  - Prije no što objekt bude uništen, poziva se metoda za finalizaciju

# 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

## ◆ Linija je poseban geometrijski lik:

```
public class Linija extends GeometrijskiLik {  
    /** X koordinata početka linije. */  
    private int x0;  
    /** Y koordinata početka linije. */  
    private int y0;  
    /** X koordinata kraja linije. */  
    private int x1;  
    /** Y koordinata kraja linije. */  
    private int y1;
```

# Java razredi

## ◆ Linija je poseban geometrijski lik:

```
/**  
 * Konstruktor linije  
 */  
public Linija(int x0, int y0, int x1, int y1) {  
    super("Linija"); // Poziv konstruktora od geom. lika  
    this.x0 = x0;  
    this.y0 = y0;  
    this.x1 = x1;  
    this.y1 = y1;  
}
```

# Java razredi

- ◆ Linija je poseban geometrijski lik:

```
/**
 * Dohvat X-koordinate početka linije
 */
public int getX0() {
    return x0;
    // isto sto i: return this.x0;
}
// ostale metode...
}
```

# 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

- ◆ 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 razred Object

- ◆ Java definira razred Object koji ima niz metoda
- ◆ Nama interesantne su:
  - `Object()`; - konstruktor bez argumenata
  - `void finalize() throws Throwable`; - finalizator
  - `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;  
}
```

# Java razredi

## ◆ Dopunimo razred Linija:

```
public boolean equals(Object obj) {  
    if( !(obj instanceof Linija) ) return false;  
    Linija druga = (Linija)obj;  
    return x0==druga.x0 && y0==druga.y0 &&  
           x1==druga.x1 && y1==druga.y1;  
}  
  
public String toString() {  
    return super.toString() + "("+x0+", " +y0+", "  
                               +x1+", " +y1+")";  
}
```

# 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+")";  
}
```



# 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
- ◆ Elipsa ima radijusX i radijusY koji su nezavisni, Kružnica treba samo jedan
- ◆ Ako Kružnica potihom mijenja oba, može pokvariti kod koji za Elipse radi

# Upravljanje pogreškama

- ◆ Što napraviti kada se u funkciji dogodi greška?
  - Prekinuti izvođenje programa
  - Vratiti status pogreške
    - ◆ Najčešće, nema posebnog statusa
    - ◆ Funkcija vraća i podatak, i status na istom mjestu → loše
    - ◆ Npr.  
`int getchar(FILE *f)`  
greška ako je rezultat manji od nule

# Upravljanje pogreškama

- ◆ Java uvodi koncept iznimke (engl. Exception)
  - ◆ Ako metoda regularno završi, sigurno vraća podatak
  - ◆ Ako se dogodi pogreška, izaziva se iznimka, i prekida se izvođenje metode
  - ◆ Iznimku netko mora uhvatiti
- 

# Upravljanje pogreškama

- ◆ Neuhvaćene iznimke rezultiraju prekidom izvođenja programa

```
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 (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

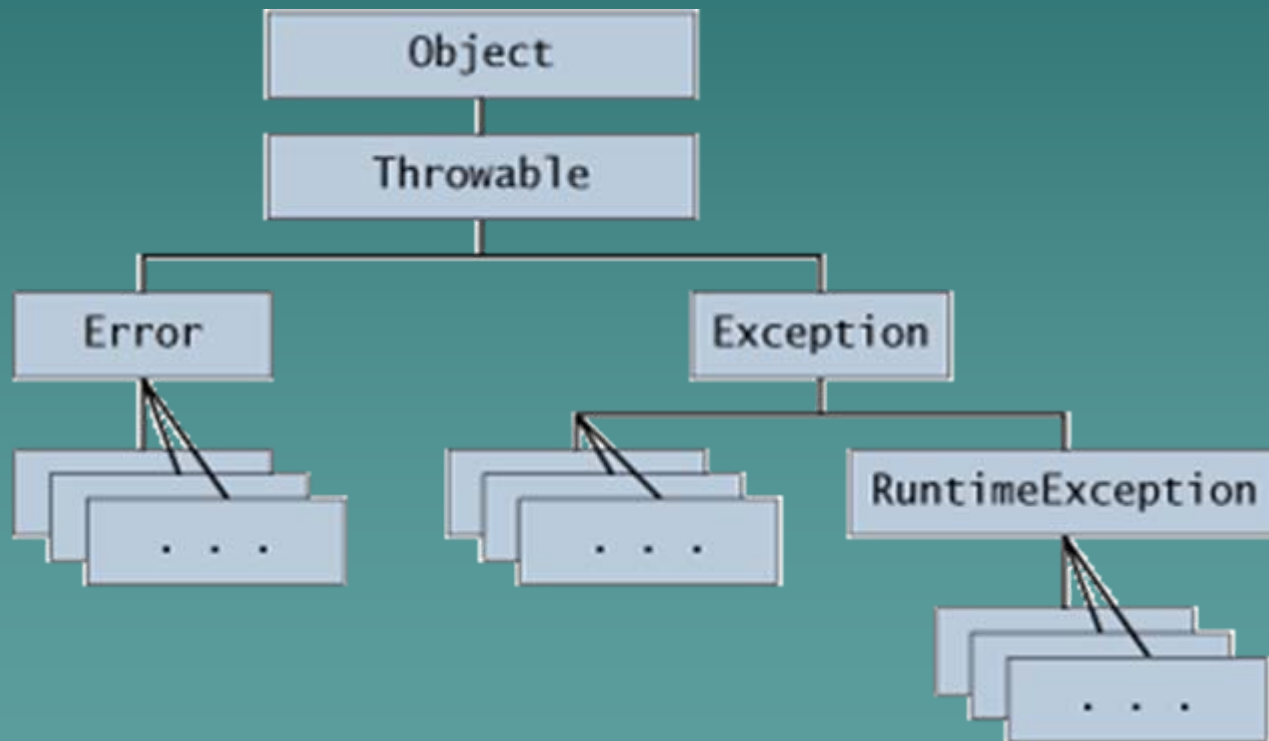
# Upravljanje pogreškama

## ◆ Besmisleni kod: krivi poredak!

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

# Upravljanje pogreškama

## ◆ Stablo nasljeđivanja iznimaka



# Upravljanje pogreškama

- ◆ 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.)
- ◆ Java 7 dodaje još neke mogućnosti



# Upravljanje pogreškama

- ◆ Pročitati:

<http://java.sun.com/docs/books/tutorial/essential/exceptions/>