Razvoj softvera

dr.sc. Emir Mešković

V predavanje

Sadržaj predavanja

- Ugniježdene klase
  - unutarnje, lokalne i anonimne klase
- Funkcionalni interfejsi
- Lambda izrazi
- Stream API

Ugniježdene (*nested*) klase

- Klasa definisana unutar druge klase

```
class VanjskaKlasa {
  ...
  class UgnijezdenaKlasa {
      ...
  }
}
```

- Ugniježdena klasa je članica klase u kojoj je definirana
  - Mogu biti deklarisane sa `private`, `public`, `protected` ili *default* modifikatorom vidljivosti (za razliku od vanjske koja može biti samo `public` ili *default*)
  - Naziva se još i unutarnja (*inner*) klasa ukoliko nije deklarsiana kao statička
  - Ima pristup članovima vanjske klase, čak i ako su deklarisani kao `private`
  - Statičke ugniježdene klase nemaju pristup članovima vanjske klase

Ugniježdene (*nested*) klase

- Razlozi za korištenje ugniježdenih klasa:
  - Način logčkog grupisanja klasa koje se koriste samo na jednom mjestu – kada je klasa korisna samo jednoj drugoj klasi
  - Povećava se enkapsulacija – ugniježdena klasa se može sakriti od vanjske upotrebe a istovremeno ima pristup privatnim članovima vanjske klase
  - Čitljiviji kod i lakši za održavanje –smješta se bliže mjestu upotrebe
- Statičke ugniježdene klase
  - Kao i statička polja i metode, pridružene su vanjskoj klasi
  - Ne mogu diektno koristiti nestatičke metode i polja, samo preko objektne reference
  - Ponašaju se kao klase najvišeg nivoa koje su ugniježdene u drugu klasu zbog praktičnosti pakiranja
  - Pristupa im se korištenjem imena vanjske klase: `VanjskaKlasa.StatickaUgnijezdenaKlasa`

Unutarnje (*inner*) klase

- Pridružena je instanci vanjske klase i ima direktan pristup metodima i poljima tog objekta
  - Samim tim ne može definirati vlastite statičke članove
- Da bi se instancirala unutarnja klasa, prvo se mora instancirati vanjska klasa
  - `VanjskaKlasa vanjskiObj = new VanjskaKlasa();`
  - `VanjskaKlasa.UnutarnjaKlasa innerObj = vanjskiObj.new UnutarnjaKlasa();`
  - Kreira se unutarni objekat unutar vanjskog objekta
- Ukoliko se deklariše član ili parametar metode u određenom opsegu vidljivosti sa istim imenom kao član iz vanjskog opsega, pristupa se lokalnijoj varijabli (*shadowing*)

Primjer inner klase


```
public class StrukturaPodataka {
    private final static int SIZE = 15;
    private int[] niz = new int[SIZE];
    public StrukturaPodataka() {
      for (int i = 0; i < SIZE; i++) { niz[i] = i; }
    }
    public void printParne() {
      StrukturaPodatakaIterator iterator = this.new ParniIterator();
      while(iterator.hasNext())
                System.out.print(iterator.next() + " ");
      System.out.println();
    }
    interface StrukturaPodatakaIterator extends java.util.Iterator<Integer> { }
    private class ParniIterator implements StrukturaPodatakaIterator {
      private int nextIndex = 0;
      public boolean hasNext() {
                return (nextIndex <= SIZE - 1);
      }
      public Integer next() {
                Integer retValue = Integer.valueOf(niz[nextIndex]);
                nextIndex += 2;
                return retValue;
      }
    }
    public static void main(String[] args) {
      StrukturaPodataka sp = new StrukturaPodataka();
      sp.printParne();
    }
```

Lokalne klase

- Klase definirane u bloku
  - Najčešće u tijelu nekog metoda, ali je moguće i u for petlji ili if naredbi
- Lokalna klasa ima pristup članovima vanjske klase i lokalnim varijablama koje su deklarisane kao final
  - Od Jave 8 može pristupiti lokalnim varijablama i parametrima koji su efektivno finalni (vrijednost se ne mijenja nakon inicijalizacije)
  - Ukoliko se pokuša promijeniti vrijednost lokalne varijable ili parametra javlja se greška pri kompajliranju
- Kao i *inner* klase ne mogu definisati statičke članove izuzev konstanti (`static final`)
- Lokalne klase u statičkim metodima mogu pristupiti samo statičkim članovima vanjske klase
- Lokalne klase nisu statičke jer imaju pristup članovima u vanjskom bloku
- Interfejsi se ne mogu definirati unutar blokova

Primjer lokalne klase


```
public class LokalnaKlasaPrimjer {
    static String regularniIzraz = "[^0-9]";

    public static void validirajTelBroj(String telBroj1, String telBroj2){
      final int duzinaBroja = 9;

      //lokalna klasa Telefonski broj
      class TelefonskiBroj {
          String formatiraniTelBroj = null;
          TelefonskiBroj(String telBroj){
                String trenutniBroj = telBroj.replaceAll(regularniIzraz, "");
                if (trenutniBroj.length() == duzinaBroja)
                    formatiraniTelBroj = trenutniBroj;
                else
                    formatiraniTelBroj = null;
          }
          public String getBroj() {
                return formatiraniTelBroj;
          }
          public void printOriginalneBrojeve() {
                System.out.println("Originalni brojevi su " + telBroj1 + " i " +
                                   telBroj2);
          }
      }
      //kraj lokalne klase
    ...
```

Primjer lokalne klase


```
        ...
        TelefonskiBroj mojBroj1 = new TelefonskiBroj(telBroj1);
        TelefonskiBroj mojBroj2 = new TelefonskiBroj(telBroj2);

        mojBroj1.printOriginalneBrojeve ();

        if (mojBroj1.getBroj() == null)
            System.out.println("Prvi broj nije validan");
        else
            System.out.println("Prvi broj je " + mojBroj1.getBroj());
        if (mojBroj2.getBroj() == null)
            System.out.println("Drugi broj nije validan");
        else
            System.out.println("Drugi broj je " + mojBroj2.getBroj());
    }
    //kraj metoda validirajTelBroj

    public static void main(String[] args) {
        validirajTelBroj("123-456-7890", "061-123-456");
    }
}
//kraj klase LokalnaKlasaPrimjer
```

Anonimne klase

- Omogućavaju istovremeno deklarisanje i instanciranje klase
  - Slične lokalnim klasama samo nemaju naziva
  - Koriste se kada je lokalnu klasu potrebno upotrijebiti samo jednom
- Anonimna klasa predstavlja izraz, što znači da je klasa definisana drugim izrazom
  - Sintaksa izraza podsjeća na poziv konstruktora, osim što postoji definicija klase u bloku koda
- Anonimna klasa se sastoji od sljedećeg:
  - `new` operatora
  - naziva interfejsa koji se implementira ili klase koja se nasljeđuje
  - zagrada unutar kojih se navode argumenti za konstruktor
  - tijela, koje predstavlja tijelo klase (deklaracije metoda ali ne i naredbe)
- Izraz koji pedstavlja definiciju anonimne klase mora biti sastavni dio neke naredbe

Anonimne klase

- Pravila pristupa za anonimne klase:
  - Mogu pristupiti članovima vanjske klase
  - Ne mogu pistupiti lokalnim varijablama koje nisu deklarisane kao final ili nisu efektivno finalne
  - Deklaracija varijable zasjenjuje deklaracije u vanjskoj klasi sa istim imenom
- Ograničenja za članove anonimnih klasa
  - Ne mogu deklarisati statičke članove ili interfejse
  - Mogu imati konstante (`static final`)
- U anonimnim klasama mogu se deklarisati polja, dodatni metodi, lokalne klase, inicijalizatore instance
- Ne mogu se deklarisati konstruktori u anonimnoj klasi
- Često se koriste u GUI aplikacijama (Swing, JavaFX)

Primjer anonimne klase


```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
class PrimjerInner extends JFrame{
    JButton button;
    public PrimjerInner(){
      setTitle("Primjer Button Action bez Lambda izraza");
      setSize(400,300);
      setVisible(true);
      setDefaultCloseOperation(EXIT_ON_CLOSE);
      button = new JButton("Klikni me");
      button.setBounds(100,100,90,40);
      button.addActionListener(new ActionListener(){
          public void actionPerformed(ActionEvent e){
              System.out.println("Upravo ste klliknuli na button.");
          }
      });
      add(button);
    }
    public static void main(String args[]){
        new PrimjerInner();
    }
}
```

Funkcionalni interfejs

- U Javi 7 - Single Abstract Method ili SAM tip
  - obično implementiran sa anonimnim klasama
- Sadrži jedan i samo jedan apstraktni metod
  - Određuje namjenu interfejsa (Runnable, Comparable, itd.)

```
import java.lang.FunctionalInterface;
@FunctionalInterface
public interface MojFuncInterfejs{
    int saberi(int a, int b);
}
```

- Anotacija je opcionalna, ali indicira da interfejs mora imati samo jedan apstraktni metod
- Olakšano pisanje koda sa anonimnim klasama
- Sintaksa još uvijek komplikovana

Lambda izraz

- Omogućuje da se ne navodi ime metoda funkcionalnog interfejsa kada se prosljeđuje kao argument
  - Mora se kreirati ili vlastiti funkcionalni interfejs ili koristiti predefinisani funkcionalni interfejs u Javi

```
@FunctionalInterface
interface MojFuncInterfejs {
   int saberi(int a, int b);
}
public class LambdaTest {
   public static void main(String args[]) {
     MojFuncInterfejs sum = (a, b) -> a + b;
     System.out.println("Result: "+sum.saberi(12, 100));
   }
}
```

- Lambda izraz ustvari predstavlja
  - Anonimni ili neimenovani metod
  - Implementaciju metoda definiranog funkcionalnim interfejsom

Lambda izraz

- Uvodi novi sintaksni element i operator u Javi
  - Lambda operator ili operator strelice (->)
- U lambda izrazu:
  - lijeva strana određuje sve parametre potrebne izrazu
  - desna strana je lambda tijelo, koje specificira akcije lambda izraza
- Dva tipa lambda izraza:
  - tijelo sa samo jednim izrazom

```
() -> System.out.println("Lambda izraz je extra");
```

  - tijelo koje se sastoji od bloka koda (uvijek se eksplicitno mora vratiti vrijednost)

```
() -> {    double pi = 3.1415;
           return pi;
}
```

- Lambda izraz sa parametrima:

```
(n) -> (n%2)==0
```

Lambda izraz

- Prvi korak ka funkcionalnom programiranju u Javi
- Anonimna funkcija koja nema naziv i ne pripada nijednoj klasi
- Jasan i koncizan način reprezentacije interfejsa metode korištenjem izraza
- Implementira funkcionalni interfejs i pojednostavljuje razvoj softvera
- Može prisupiti vanjskim varijablama pod određenim uslovima:
  - Mogu se deklarisati lokalne varijble u lambda izrazu koje nemaju isti naziv kao vanjske varijable – ne uvodi novi nivo vidljivosti
  - Može pristupiti lokalnim varijablama i parametrima vanjskog bloka koji su final ili efektivno final
  - Može pistupiti statičkom članu klase

Reference metoda

- Skraćena notacija lambda izraza za poziv metoda
- Npr. lambda izraz:

```
str -> System.out.println(str)
```

- se može zamijeniti sa referencom metoda:

```
System.out::println
```

- Operator :: se koristi u referenci metoda za razdvajanje klase ili objekta od naziva metoda
- Postoje 4 tipa reference metoda:
  - referenca metoda na metod instance objekta - `object::instanceMethod`
  - referenca metoda na statički metod klase - `Class::staticMethod`
  - referenca metoda na metod instance proizvoljnog objekta određenog tipa - `Class::instanceMethod`
  - referenca metoda na konstruktor - `Class::new`

Reference metoda

- Referenca na statički metod:

```
public interface Display {
    public int show(String s1, String s2);
}
public class Test {
    public static int doShow(String s1, String s2){
        return s1.lastIndexOf(s2);
    }
}
Display disp = Test::doShow;
```

- Referenca na metod instance proizvoljnog objekta određenog tipa

```
Display disp = String::indexOf;
```

Reference metoda

- Referenca na metod instance objekta

```
public interface Deserializer {
     public int deserialize(String v1);
}
public class StringConverter {
     public int convertToInt(String v1){
          return Integer.valueOf(v1);
     }
}
stringConverter strConv = new StringConverter();
Deserializer deserializer = strConv::convertToInt;
```

- Referenca na konstruktor

```
public interface Factory {
     public String create(char[] val);
}
Factory fact = String::new;
```

Java Stream API

- Nalazi se u `java.util.stream` paketu
- Omogućava izvođenje različitih agregatnih operacija nad podacima kolekcija, nizova, I/O operacija
- Način korištenja streamova u Javi:
  - kreirati stream
  - izvesti međuoperacije na početnom streamu kako bi se transformisao u drugi stream i nastaviti sa drugim međuoperacijama.
  - izvesti konačnu (terminirajuću) operaciju na konačnom streamu kako bi se dobio rezultat.
- Osobine Java streamova
  - stream ne pohranjuje elemente. Jednostavno izvodi agregatne operacije kako bi se dobio željeni stream podataka
  - agregatne operacije ne mijenjaju izvorne podatke nego jednostavno vraćaju novi stream.
  - Sve stream operacije su po prirodi *lazy* što znači da se ne izvršavaju dok nisu potrebne.

Java Stream vs Collection

- Kolekcije su strukture podataka pohranjene u memoriji
  - Svaki element kolekcije mora biti izračunat prije dodavanja u kolekciju
- Stream je konceptualno fiksna struktura podataka u kojoj se elementi izračunavaju na zahtjev
- Rezultat međuoperacija je stream što omogućava ulančavanje više operacija (poziva metoda) – *pipelining*
- Stremovi su dizajnirani za korištenje lambda izraza
- Ne podržavaju pristup korištenjem indeksa
- Mogu se pretvoriti u niz ili listu
- Omogućeno sekvencijalno i paralelno izvršavanje operacija

Kreiranje stream-a

- Korištenjem `of()` statičkog metoda klase Stream
  - `Stream<Integer> stream = Stream.of(1,2,3,4,5,6,7,8,9);`
  - `Stream<Integer> stream = Stream.of(new Integer[] {1,2,3,4,5,6,7,8,9});`
- Korištenjem `stream()` metoda Collection interfejsa
  - `List<Integer> list = new ArrayList<Integer>();`
  - `Stream<Integer> stream = list.stream();`
- Korištenjem `generate()` i `iterate()` statičkih metoda Stream klase
  - `Stream<Date> stream = Stream.generate(() -> { return new Date(); });`
- Iz stringa karaktera ili stringa tokena
  - `IntStream stream = "12345_abcdefg".chars();`
  - `Stream<String> stream = Stream.of("A$B$C".split("\\$"));`

Konverzija stream-a u kolekciju

- Konverzija stream-a u listu
  - `List<Integer> listaParnih = stream.filter(i -> i%2 == 0) .collect(Collectors.toList());`
- Konverzija stream-a u niz
  - `Integer[] nizParnih = stream.filter(i -> i%2 == 0). toArray(Integer[]::new);`
- Kako bi se omogućio paralelizam kreirati paralelni stream umjesto sekvencijalnog
  - Koristiti `parallelStream()` metod umjesto `stream()` metoda

Java Stream API

- `filter()` čita podatke iz streama i vraća novi stream nakon transformisanja podataka na osnovu datog uslova
- `map()` konvertuje elemnte streama primjenom posebne funkcije nad njima i prikljuplja nove elemente u stream
- `sorted()` vraća sortirani stream
- `concat()` spaja dva streama
- `collect()` metod prikuplja konačni stream i konvertuje ga u kolekciju
- forEach() iterira kroz sve elemente streama i izvodi zadanu operaciju
- `distinct()` kreira novi stream jedinstvenih elemenata
- `count()` vraća veličinu streama
- `anyMatch()`, `allMatch()`, `noneMatch()` validira elemente streama u odnosu na neki predikat
- `reduce()` reducira stream elemenata na određenu vrijednost pomoću funkcije koja se proslijedi

Java Stream API


```
import java.util.ArrayList;
import java.util.List;
public class StreamExample{
   public static void main(String[] args) {
       List<String> imena = new ArrayList<String>();
       imena.add("Damir");
       imena.add("Dino");
       imena.add("Maja");
       imena.add("Deni");
       imena.add("Ema");
       imena.add("Goran");
       imena.add("Lejla");
       long count = imena.stream()
                    .filter(str->str.length()<5).count();
       System.out.println("Postoji "+ count + " imena čija je
                           dužina manja od 5");
   }
}
```