Lambda izrazi

Programski jezici II

Funkcionalno programiranje

- Korišćenjem funkcionalnog programiranja mnoge klase problema se mogu jednostavnije riješiti nego objektno-orijentisanim pristupom
  - takav programski kod je često „jasniji“ i jednostavniji za održavanje
- Funkcionalno programiranje u Java programskom jeziku ne zamjenjuje objektno-orijentisano programiranje, ali proširuje i unapređuje mnoge metode i algoritme

Prosljeđivanje funkcija

- Mnogi programski jezici dozvoljavaju prosljeđivanje funkcija kao parametara metoda
  - Dinamički i obično slabo tipizirani:
    - JavaScript, Lisp, Scheme,...
  - Snažno tipizirani
    - Ruby, Scala, Clojure, ML,...

Lambda izrazi

- Lambda izrazi omogućavaju jednostavan način rada sa metodama anonimnih objekata anonimnih lokalnih klasa koje implementiraju interfejse sa samo jednom metodom
- Lambda izrazi nemaju (eksplicitno) definisanu klasu kojoj pripadaju što ih čini sličnim globalnim funkcijama kakve postoje u nekim drugim programskim jezicima
- S druge strane, postoji automatski generisana klasa „omotač“ funkcije i njen automatski instanciran objekat – ova klasa je lokalna
- Dakle, na mjestu definisanja lambda izraza kreira se objekat anonimne klase koja implementira interfejs sa jednom apstraktnom metodom – ovakvi interfejsi se nazivaju funkcionalni interfejsi ili SAM (*Single Abstract Method*) interfejsi

Lambda izrazi

- Koncizna sintaksa
  - Jezgrovitiji i čistiji pristup u poređenju sa anonimnim unutrašnjim klasama
- Nedostaci anonimnih unutrašnjih klasa
  - Glomazne, konfuzija pri korišćenju ključne riječi „this“, nema pristupa lokalnim promjenjivim koje nisu final, teško za optimizovati, ...
- Pogodne za korišćenje sa Stream API-jem
- Programerima je koncept poznat
  - Callback, map/reduce idiom, ...

Lambda izrazi

- „Klasični“ pristup

```
button1.addActionListener(new ActionListener() {
      @Override
      public void actionPerformed(ActionEvent event) {
        setBg(Color.BLUE);
      }
});
```

- „Novi“ pristup

```
button1.addActionListener(event -> setBg(Color.BLUE));
```

Lambda izrazi

- Ohrabruje se upotreba funcionalnog programiranja
  - Mnoge klase problema se lakše rješavaju upotrebom funkcionalnog programiranja i rezultiraju u programskom kodu koji je jasniji i jednostavniji za održavanje
- Funkcionalno programiranje ne zamjenjuje OOP – OOP je i dalje osnovni pristup za predstavljanje tipova, dok funkcionalno programiranje može poboljšati mnoge metode i algoritme
- Podržava se upotreba stream-ove
  - Stream-ovi su wrapper-i oko data source-va (nizovi, kolekcije, itd.) koji koriste lambde, podržavaju map/filter/reduce, koriste lazy evaluaciju i mogu se učiti paralelnim (automatski)
  - „Klasični“ pristup

```
for(Employee e: employees) {
    e.doSomething();
}
```

  - „Novi“ pristup

```
employees.stream().parallel().forEach(e -> e.doSomething());
```

Lambda izrazi

- Java 7 primjer

```
Arrays.sort(testStrings, new Comparator<String>() {
        @Override
        public int compare(String s1, String s2) {
                return(s1.length() - s2.length());
        }
});
```

- Java 8 primjer

```
Arrays.sort(testStrings,
(s1, s2) -> s1.length() - s2.length());
```

Lambda izrazi

- Java 7 primjer

```
Arrays.sort(testStrings, new Comparator<String>() {
          @Override
          public int compare(String s1, String s2) {
                    return(s1.length() - s2.length());
          }
});
```

- Java 8 primjer

```
Arrays.sort(testStrings,
(String s1, String s2) -> {return (s1.length() - s2.length()); });
```

- Ideja
  - Iz API-ja, Java „zna“ da je Comparator drugi argument metode Arrays.sort, tako da to nije potrebno naglašavati
  - Comparator ima samo jednu metodu, tako da nije potrebno naglašavati naziv metode – compare
  - Dodaje se “->” između parametara metode i tijela metode

Lambda izrazi

- Java 7 primjer

```
Arrays.sort(testStrings, new Comparator<String>() {
        @Override
        public int compare(String s1, String s2) {
                return(s1.length() - s2.length());
        }
});
```

- Java 8 primjer

```
Arrays.sort(testStrings,
(s1, s2) -> {return (s1.length() - s2.length()); });
```

- Ideja
  - Gledajući prvi argument metode sort (testStrings), Java može zaključiti da je tip drugog argumenta Comparator<String>, te da su, prema tome, parametri compare metode String-ovi – iz tog razloga nije potrebno naglašavati tipove parametara metode compare
  - Java i dalje radi snažnu provjeru tipova u vrijeme kompajliranja – razlika je u tome što kompajler zaključuje o kojim tipovima se radi – slično kao sa diamond operatorom

    ```
    List<String> abc = new ArrayList<>();
    ```

  - U slučaju da su tipovi dvosmisleni, kompajler će prijaviti da ne može zaključiti o kojim tipovima se radi – u ovom slučaju tipove je potrebno navoditi

Lambda izrazi

- Java 7 primjer

```
Arrays.sort(testStrings, new Comparator<String>() {
          @Override
          public int compare(String s1, String s2) {
                    return(s1.length() - s2.length());
          }
});
```

- Java 8 primjer

```
Arrays.sort(testStrings, (s1, s2) -> s1.length() - s2.length());
```

- Ideja
  - Ako tijelo metode čini jedan return izraz, vitičaste zagrade i „return“ mogu da se odbace
  - Ove se ne može uvijek uraditi, pogotovo ako se koriste petlje ili if naredbe
  - Lambde se obično koriste kada je tijelo metode kratko, tako da je ovo obično moguće uraditi

Lambda izrazi

- Ako metoda interfejsa ima tačno jedan parametar i male zagrade je moguće odbaciti
- Java 7 primjer

```
button.addActionListener(new ActionListener() {
        @Override
        public void actionPerformed(ActionEvent e) {
                doSomethingWith(e);
        }
});
```

- Java 8 primjer – sa malim zagradama

```
button.addActionListener((e) -> doSomethingWith(e));
```

- Java 8 primjer – bez malih zagrada

```
button.addActionListener(e -> doSomethingWith(e));
```

- U praksi – stvara se objekat anonimne klase koja implementira interfejs sa jednom metodom

Funkcionalni interfejsi

- Funkcionalni interfejsi – interfejsi koji imaju jednu metodu
  - Comparator, Runnable, itd.
- Drugi naziv – SAM (Single Abstract Method) interfejsi
- @FunctionalInterface
  - Detektuje greške za vrijeme kompajliranja
    - Ako se naknadno doda druga apstraktna metoda u interfejs, interfejs se neće moći kompajlirati
  - Izražava namjeru predviđenu dizajnom
    - „govori“ programerima da će se za ovaj interfejs koristiti lambde
  - Nije obavezan
    - Isto kao i @Override
    - Lambde se mogu koristiti svuda gdje se očekuju interfejsi sa jednom apstraktnom metodom (funkcionalni interfejsi, SAM interfejsi), bez obzira da li taj interfejs koristi @FunctionalInterface

Funkcionalni interfejsi

- Funkcionalni interfejsi – interfejsi koji imaju jednu metodu
- U slučaju da interfejs deklariše apstraktnu metodu koja redefiniše jednu od javnih metoda klase Object, ovakav interfejs će se i dalje smatrati za funkcionalni, jer će bilo koja klasa koja implementira ovakav interfejs imati implementaciju date metode nasljeđenu iz klase Object
  - U ove metode se ubrajaju: toString, equals i hashCode
- Ovo je slično situaciji u kojoj interfejs ima jednu apstraktnu metodu i jednu ili više podrazumijevanih (default) metoda, koji se, takođe, smatra za funkcionalni interfejs
- Interfejsi koji nasljeđuju funkcionalni intefejs, ali ne dodaju apstraktne metode takođe su funkcionalni
- Lambda izraz se može pojaviti na svim mjestima u kodu na kojim se očekuje objekat klase koja implementira funkcionalni interfejs

```
public interface NumberInterface {
    int getNumber();
}
```

```
public class Test {
    public static void main(String[] args) {
        NumberInterface ni;
        ni = () -> 11;
        System.out.println(ni.getNumber());
    }
}
```

Funkcionalni interfejsi

- Kada se lambda izraz konvertuje u instancu funkcionalnog interfejsa, potrebno je obratiti pažnju na provjerene izuzetke – ako tijelo lambda izraza može baciti provjereni izuzetak, onda to mora biti i naglašeno u apstraktnoj metodi ciljnog interfejsa

```
Runnable r = () -> {
    Thread.sleep(1);
    int sum = 0;
    for (int i = 0; i < 100; i++) {
        sum += i;
    }
    System.out.println(sum);
};
```

- pokušaj kompajliranja ovog koda dovešće do greške, jer metoda sleep klase Thread može baciti provjereni izuzetak InterruptedException, a metoda run interfejsa Runnable takav izuzetak (niti bilo koji drugi) ne može baciti.

Funkcionalni interfejsi

- da bi se prethodni kod mogao kompajlirati potrebno je:
  - uhvatiti InterruptedException izuzetak u tijelu lambda izraza ili
  - dodijeliti ovaj lambda izraz promjenljivoj koja je tipa interfejsa čija apstraktna metoda može baciti izuzetak tipa InterruptedException
    - npr. Callable<Void>

Funkcionalni interfejsi u java.util.function paketu

- Java API posjeduje određen broj često korišćenih funkcionalnih interfejsa koji su smješteni u java.util.function paketu
- ovi interfejsi su funkcionalni interfejsi opšte namjene
- svi interfejsi u ovom paketu su anotirani @FunctionalInterface anotacijom

Funkcionalni interfejsi u java.util.function paketu

- postoje četiri osnovna tipa funkcija koji su predstavljeni odgovarajućim funkcionalnim interfejsima:
  - Function<T,R> - predstavlja funkciju koja prihvata jedan argument tipa T i vraća rezultat tipa R
    - R apply(T t)
  - Predicate<T> - predstavlja funkciju koja prihvata jedan argument tipa T i vraća rezultat tipa boolean
    - boolean test(T t)
  - Consumer<T> - predstavlja funkciju koja prihvata jedan argument tipa T i ne vraća rezultat, tj. povratni tip je void
    - void accept(T t)
  - Supplier<T> - predstavlja funkciju koja nema argumente i vraća rezultat tipa T
    - T get()

Funkcionalni interfejsi u java.util.function paketu

- postoje varijante ovih funkcija (interfejsa) koje rade sa dva parametra:
  - BiFunction<T,U,R> - predstavlja funkciju koja prihvata dva argumenta tipa T i U i vraća rezultat tipa R
    - R apply(T t, U u)
  - BiPredicate<T,U> - predstavlja funkciju koja prihvata dva argumenta tipa T i U i vraća rezultat tipa boolean
    - boolean test(T t, U u)
  - BiConsumer<T,U> - predstavlja funkciju koja prihvata dva argumenta tipa T i U i ne vraća rezultat
    - void accept(T t, U u)

Funkcionalni interfejsi u java.util.function paketu

- postoje i funkcionalni interfejsi koji proširuju funkcionalnosti osnovnih funkcionalnih interfejsa:
  - UnaryOperator<T> - predstavlja operaciju nad jednim operandom koja vraća rezultat istog tipa kao što je tip operanda – specijalizacija Function funkcije koja radi sa argumentom tipa T i vraća rezultat tipa T
  - BinaryOperator<T> - predstavlja operaciju nad dva operanda istog tipa koja vraća rezultat istog tipa kao što je tip operanada – specijalizacija BiFunction funkcije koja prihvata dva argumenta tipa T i vraća rezultat tipa T

Funkcionalni interfejsi u java.util.function paketu

- postoje i funkcionalni interfejsi koji predstavljaju specijalizaciju drugih interfejsa na takav način da su tipske promjenljive zamijenjene konkretnim tipovima su:
  - BooleanSupplier - specijalizacija Supplier funkcije koja vraća boolean vrijednost
  - DoubleBinaryOperator - specijalizacija BinaryOperator funkcije koja radi sa operandima tipa double i vraća rezultat tipa double
  - DoubleConsumer - specijalizacija Consumer funkcije koja radi sa argumentom tipa double
  - …

Lambda izrazi

- Opšti oblik definicije lambda izraza: (parametri) -> {tijelo}
  - parametri – uobičajena lista tipova i imena
    - tipovi mogu da se izostave, ako se o njima može zaključiti iz konteksta
    - moraju se ili navesti svi tipovi ili izostaviti svi tipovi
    - ako u listi postoji samo 1 parametar bez tipa – zagrade () mogu da se izostave
    - zagrade () su obavezne ako nema parametara
  - tijelo – uobičajeno tijelo funkcije (blok koji koristi parametre)
    - ako je samo jedna naredba (return), mogu da se izostave zagrade {} i ključna riječ return
    - tada se tijelo svodi na izraz koji izračunava vrijednost lambda izraza
  - tip lambda izraza – ne navodi se
    - o njemu se zaključuje na osnovu tipa izraza u return naredbi
    - tip može biti i void

Reference metoda

- Često se javlja potreba da se pozove metoda nad argumentom lambda izraza
  - variable -> variable.getMethod()
  - variable::getMethod
- Osnovna ideja jeste koristiti referencu metode kako bi se neka postojeća metoda mogla koristiti i tretirati kao lambda izraz
- Standardni oblici reference metoda:

```
object::instanceMethod()
Class::staticMethod()
Class::instanceMethod()
```

Vidljivost varijabli u Lambda izrazima

- Lambde ne uključuju novi nivo vidljivosti
  - this promjenljiva referencira vanjsku klasu, ne anonimnu unutrašnju klasu u koju je Lambda uključena
  - Ne postoji “OuterClass.this” promjenjiva, osim ako se Lambda ne nalazi u običnoj unutrašnjoj klasi
  - Ne mogu „uvesti“ ime nove promjenjive koje je identično imenu promjenjive u metodi koja kreira lambdu
  - Lambde mogu referencirati (ne i modifikovati) lokalne varijable iz okružujuće metode
    - I dalje mogu referencirati i modifikovati promjenjive instance iz okružujuće klase