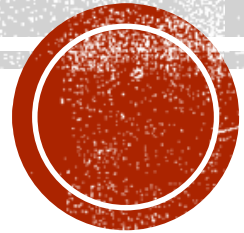
ОБЈЕКТНО ОРЈЕНТИСАНО ПРОГРАМИРАЊЕ

ПРОГРАМСКИ ЈЕЗИК ЈАВА — 2

Ламбда изрази



ЛАМБДА ИЗРАЗИ И ФУНКЦИОНАЛНА ПАРАДИГМА

- Ламбда изрази нису изворно елементи објектне-оријентисане парадигме.
  - Они се везују за тзв. функционалну парадигму у којој је све функцијски објекат и где се сви проблеми решавају прављењем одговарајућих композиција функција и применом рекурзије.
- Неки од познатијих функционалних програмских језика су: Lisp, Haskell, F#, Elrang и други.
- Већина модерних објектно-оријентисаних језика је прихватила и интегрисала функционалну парадигму.
  - Ово је омогућило пре свега лакши рад са колекцијама и токовима података
  - Због своје декларативне природе, ламбда изрази омогућавају програмеру интуитивније изражавање, те се очекује да у будућности буду незаобилазни део Јава, али и других ОО програмских језика.

МОТИВАЦИЈА

- Употреба анонимних класа је обично везана за *ad-hoc* имплементацију одређене функције:
  - Ова функција се користи једнократно;
  - Смисао постојања овакве класе је пренос функције, па се често он назива и функцијски објекат (слично показивачу на функцију);
  - Овакав запис је, међутим, доста гломазан.
- Ламбда изрази омогућавају да се ова иста употреба реализује елегантније.
- Кроз наредних неколико примера ће, корак по корак, бити разјашњена ова мотивација.

ПРИМЕР – ОПИС ПРОБЛЕМА

- Претпоставимо да је потребно да дизајнирамо **социјалну мрежу**.
- Једна од могућности у оквиру система је административни панел који би омогућио:
  - Администратору да излиста све кориснике
    који испуњавају одређени критеријум;
  - На овај начин би администратор даље могао да врши одређене активности над тим корисницима:
    - Шаље поруке и обавештења;
    - Брише или ажурира податке и слично.
  - Критеријума може да буде доста,
    па је потребно водити рачуна о компактности кода.

ПРИМЕР – ЕНТИТЕТ

Главни ентитет је класа која представља особу:

```
public class Person {
        public enum Sex { MALE, FEMALE }
        String name;
        LocalDate birthday;
        Sex gender;
        String emailAddress;

        public int getAge() {
                // ...
        }

        public void printPerson() {
                // ...
        }
}
```

ПРИМЕР – НАИВНИ ПРИСТУП – ПОЈЕДИНАЧНЕ ФУНКЦИЈЕ

Нека је први захтев листање свих корисника који су старији од задатог броја година:

```
static void printPersonsOlderThan(List<Person> roster, int age) {
        for (Person p : roster) {
                if (p.getAge() >= age) {
                        p.printPerson();
                }
        }
}
```

ПРИМЕР – НАИВНИ ПРИСТУП – ПОЈЕДИНАЧНЕ ФУНКЦИЈЕ (2)

Међутим, пожељно је да излистамо и особе које су у неком задатом опсегу година:

```
static void printPersonsWithinAgeRange( List<Person> roster, int low, int high) {
        for (Person p : roster) {
                if (low <= p.getAge() && p.getAge() < high) {
                        p.printPerson();
                }
        }
}
```

Јасно је да критеријума може да буде јако пуно, па прављење појединачних функција за сваки критеријум није решење!

ПРИМЕР – НАПРЕДНИЈИ ПРИСТУП – АНОНИМНЕ КЛАСЕ

Следећи логичан корак би било прављење методе за испис особа којој се жељени критеријум прослеђује као функцијски објекат, односно анонимна класа.

```
static void printPersons( List<Person> roster, CheckPerson tester) {
        for (Person p : roster) {
                if (tester.test(p)) {
                        p.printPerson();
                }
        }
}
```

Прослеђени функцијски објекат за задавање критеријума имплментира интерфејс `CheckPerson`, који има следећу структуру:

```
interface CheckPerson {
        boolean test(Person p);
}
```

ПРИМЕР – НАПРЕДНИЈИ ПРИСТУП – АНОНИМНЕ КЛАСЕ (2)

Тада је, на доста елегантнији начин, могуће задати доста произвољан критеријум, нпр:

```
printPersons( roster,
              new CheckPerson() {
                  public boolean test(Person p) {
                      return p.getGender() == Person.Sex.MALE && p.getAge() >= 18
                             && p.getAge() <= 25;
                  }
              }
);
```

ПРИМЕР – ЈОШ НАПРЕДНИЈИ ПРИСТУП – ЛАМБДА ИЗРАЗИ

Међутим, употребом ламбда израза то може да се изрази на још елегантнији начин:

```
printPersons( roster,
              (Person p) -> p.getGender() == Person.Sex.MALE && p.getAge() >= 18
                            && p.getAge() <= 25
);
```

СИНТАКСА ЛАМБДА ИЗРАЗА

Ламбда изрази се састоје од три дела:

1. `(arg1, arg2, ...)`

   листа параметара ламбда израза;

2. `->`

   симбол који листу параметара раздваја од тела ламбда израза;

3. тело ламбда израза

   који по синтакси представља валидан Јава израз, и чија повратна вредност мора да одговара повратном типу наведеном у одговарајућем интерфејсу.

Типови параметара и тип резулата ламбда израза морају бити у сагласности са типовима који су дефинисани у одговрајућем методу интерфејса који дати ламбда израз реализује (имплементира).

ФУНКЦИОНАЛНИ ИНТЕРФЕЈСИ

Функционални интерфејси у Јави омогућавају програмирање у складу са парадигмом функционалног програмирања, где се програм третира као израчунавање математичких функција и где се избегавају стања и променљиви податци.

Функционални интерфејси, заједно са ламбда изразима и референцама на методе омогућавају писање читљивијег и чистијег програмског кода.

Функционални интерфејс је интерфејс који садржи тачно један апстракни метод. Другим речима, функционални интерфејси излажу тачно једну функцију.

Почев од Јава 8, примерци функционалног интерфејса се могу представити ламбда изразом.

Иако функционални интерфејс садржи тачно једам апстрактан метод, он може садравати већи прој подразумеваних метода.

Примери функционалних интерфејса су интерфејси `Runnable`, `ActionListener` и `Comparable`.

Функционални интерфејси се по правилу означавају анотацијом `@FunctionalInterface`.

УГРАЂЕНИ ФУНКЦИОНАЛНИ ИНТЕРФЕЈС `Predicate`

Уместо да сами дефинишемо интерфејсе, могуће је користити неке стандардне уграђене генеричке:

```
static void printPersonsWithPredicate( List<Person> roster, Predicate<Person> tester) {
        for (Person p : roster) {
                if (tester.test(p)) {
                        p.printPerson();
                }
        }
}
```

Овде је `Predicate<T>` уграђени генерички интерфејс из пакета `java.util.function`:

```
interface Predicate<T> {
        boolean test(T t);
         default Predicate<T> and(Predicate<? super T> other);
         default Predicate<T> negate();
         default Predicate<T> or(Predicate<? super T> other);
         static <T> Predicate<T> isEqual(Object targetRef);
         static <T> Predicate<T> not(Predicate<? super T> target);
}
```

УГРАЂЕНИ ФУНКЦИОНАЛНИ ИНТЕРФЕЈС `Consumer`

Генерички интерфејс `Consumer<T>` је такође предефинисан и он се налази у пакету `java.util.function`:

```
interface Consumer<T> {
        void accept(T t);
        default Consumer<T> andThen(Consumer<? super T> after){
           …
        }
}
```

Овај интерфејс представља операцију која прихвата један улазни аргумент и не враћа никакав резултат.

За разлику од осталих функционалних интерфејса, за интерфејс `Consumer` се и очекује да треба да оперише помоћу бочних ефеката.

УГРАЂЕНИ ФУНКЦИОНАЛНИ ИНТЕРФЕЈС `Supplier`

Генерички интерфејс `Supplier<T>` представља снадбевача за резултат:

```
interface Supplier<T> {
        T get();
}
```

Не постоје никакви додтни захтеви да нови или другачији резултат треба да буде враћен сваки пут када је позван снадбевач.

УГРАЂЕНИ ФУНКЦИОНАЛНИ ИНТЕРФЕЈС `Function`

Генерички интерфејс `Function<T, R>` представља функцију са једним аргументом која враће резултат.

Овај интерфејс се најчешће користи ради мапирања (пресликавања) вредности приликом рада са токовима.

Интерфејс `Function<T, R>` садржи четири метода:

```
interface Function<T, R> {
        R apply(T var1);
        default <V> Function<V, R> compose(Function<V, T> before);
        default <V> Function<T, V> andThen(Function<R, V> after);
        static <T> Function<T, T> identity();
}
```

УГРАЂЕНИ ФУНКЦИОНАЛНИ ИНТЕРФЕЈС `BiFunction`

Генерички интерфејс `BiFunction<T, U, R>` представља функцију са два аргумента која враће резултат. Три параметра типа у овом интерфејсу су:

- `T`: означава тип првог аргумента функције
- `U`: означава тип другог аргумента функције
- `R`: означава типповратне вредности тј. резултата функције

Ламбда израз који се додељује објекту типа `BiFunction` служи за дефиницију `apply` метода и описује како се добија резулата примене ове функције на дате аргументе.

Главна предност интерфејса `BiFunction` је то што он допушта коришћење два улазна аргумента, за разилу од интерфејса `Function` који је очекивао један улазни аргумент.

УГРАЂЕНИ ФУНКЦИОНАЛНИ ИНТЕРФЕЈС `BinaryOperator`

Генерички интерфејс `BinaryOperator<T>` представља бинарни оператор који на основу вредности два операнда генерише резултат.

Овај интерфејс се разликује од BiFunciton интерфејса тиме што код њега и оба операнда и резултат морају бити истог типа.

Овај функционални интерфејс има само један генерики параметар типа:

- `T`: означава тип оба улазна аргумента и тип резултата

Интерфејс `BinaryOperator<T>` проширује интерфејс `BiFunction<T, T, T>`. Најважнији (и једини апстрактни) метод код `BiFunction` интерфејса је метод `apply(T t, T u)`.

Према томе, ламбда израз који представља објекат типа `BinaryOperator` ће описивати реализацију `apply` метода, којим се дати бинарни оператор примењује над операндима.

ПРИМЕР – stream, filter, forEach

Постоји и нешто погоднија нотација за примену ламбда израза над колекцијама података:

```
roster
    .stream()
    .filter( p -> p.getGender() == Person.Sex.MALE && p.getAge() >= 18
                                  && p.getAge() <= 25)
    .forEach(p -> System.out.println(p));
```

Где су функције `stream`, `filter` и `forEach` дефинисане на следећи начин:

`Stream<E> stream()`	Претвара произвољну колекцију у ток за рад са ламбда изразима
`Stream<T> filter(` `  Predicate<? super T> predicate)`	Примењује операцију `Predicate<? super T>` над свим елементима тока
`void forEach(` `  Consumer<? super T> action)`	Извршава `void` функцију над свим елементима тока, који су преостали након филтрирања (`filter`)

ПРИМЕР – `filter`, `map`, `forEach`

Поред могућности филтрирања колекције и примене **void** функције, могуће су и трансформације сваког елемента листе у други тип података:

```
roster
    .stream()
    .filter( p -> p.getGender() == Person.Sex.MALE && p.getAge() >= 18 && p.getAge() <= 25)
    .map(p -> p.getName().length())
    .forEach(len -> System.out.println(len));
```

- Функцијом `map` се од сваке појединачне особе узима дужина имена, а потом се у `forEach` функцији даље подразумевано ради са тим податком, дакле, целим бројем
- Овај број се локално именује са `len`, а могло је и било којим другим именом

ПРИМЕР – `filter`, `map`, `sum`

Досад је размотрена само примена **void** функције над елементима колекције, међутим, могуће је да као резултат неких операција над колекцијом буде враћен резултат:

```
roster
    .stream()
    .filter( p -> p.getGender() == Person.Sex.MALE && p.getAge() >= 18
                && p.getAge() <= 25)
    .map(p -> p.getName().length())
    .sum();
```

- У овом примеру се над дужинама имена особа примењује сумирање, такорећи, рачуна се сума дужина имена особа које су мушкарци између 18 и 25 година;
- Могуће је извршавати и разне друге произвољне агрегације над колекцијом употребом функције `reduce`.