# Programiranje 1

## Poglavje 11: Lambde

Luka Fürst

# Urejanje seznama po različnih kriterijih

- seznam objektov tipa `Oseba` (`List<Oseba>`) želimo urejati po različnih kriterijih
- uporabimo metodo `sort`, ki sprejme primerjalnik (`Comparator`)
- primerjalnik ustvarimo kot objekt razreda, ki implementira vmesnik `Comparator`

# Razredi za posamezne kriterije

- razred za primerjavo po priimku

```
public class Oseba {
    ...
    private static class PrimerjalnikPoPriimku
            implements Comparator<Oseba> {

        @Override
        public int compare(Oseba prva, Oseba druga) {
            return prva.priimek.compareTo(druga.priimek);
        }
    }

    public static Comparator<Oseba> poPriimku() {
        return new PrimerjalnikPoPriimku();
    }
}
```

# Razredi za posamezne kriterije

- razred za primerjavo po spolu in starosti

```
public class Oseba {
    ...
    private static class PrimerjalnikPoSpoluInStarosti
            implements Comparator<Oseba> {

        @Override
        public int compare(Oseba prva, Oseba druga) {
            if (prva.spol != druga.spol) {
                return druga.spol - prva.spol;
            }
            return prva.letoRojstva - druga.letoRojstva;
        }
    }

    public static Comparator<Oseba> poSpoluInStarosti() {
        return new PrimerjalnikPoSpoluInStarosti();
    }
}
```

# Klic metode za urejanje

```
List<Oseba> osebe = new ArrayList<>(List.of(
    new Oseba("Jože", "Gorišek", 'M', 1956),
    new Oseba("Marija", "Gorišek", 'Z', 1959),
    ...
));
...
osebe.sort(Oseba.poPriimku());
...
osebe.sort(Oseba.poSpoluInStarosti());
...
```

# Anonimni notranji razred

- razred brez imena, definiran znotraj metode kot podrazred nekega razreda ali kot implementacijski razred za nek vmesnik
- skupaj z razredom ustvarimo tudi njegov objekt

```
R objekt = new R() {
    (re)definicije metod
};
```

- korak 1: ustvari se podrazred razreda `R`
- korak 2: izdela se objekt tega podrazreda
- omejitev: konstruktorja ne moremo definirati
  - uporabi se privzeti konstruktor, ki ne naredi ničesar

# Anonimni notranji razred

- namesto

```
public static Comparator<Oseba> primerjalnikPoPriimku() {
    return new PrimerjalnikPoPriimku();
}
private static class PrimerjalnikPoPriimku implements Comparator<Oseba> {
    @Override
    public int compare(Oseba prva, Oseba druga) {
        return prva.priimek.compareTo(druga.priimek);
    }
}
```

lahko pišemo

```
public static Comparator<Oseba> primerjalnikPoPriimku() {
    return new Comparator<Oseba>() {
        @Override
        public int compare(Oseba prva, Oseba druga) {
            return prva.priimek.compareTo(druga.priimek);
        }
    };
}
```

# Funkcijski vmesnik

- vmesnik z eno samo abstraktno metodo
- nekaj jih že poznamo ...

```
public interface Comparable<T> {
    public abstract int compareTo(T drugi);
}

public interface Comparator<T> {
    public abstract int compare(T prvi, T drugi);
}

public interface Iterable<T> {
    public abstract Iterator<T> iterator();
}
```

- vmesnik `Iterator` ni funkcijski vmesnik, saj vsebuje dve abstraktni metodi (`hasNext` in `next`)

# Funkcijski vmesnik

- paket `java.util.function` vsebuje veliko funkcijskih vmesnikov

```
public interface Predicate<T> {
    public abstract boolean test(T t);
}
public interface Consumer<T> {
    public abstract void accept(T t);
}
public interface Supplier<T> {
    public abstract T get();
}
public interface Function<T, R> {
    public abstract R apply(T t);
}
public interface BiFunction<T, U, R> {
    public abstract R apply(T t, U u);
}
...
```

# Lambda

- kadar želimo implementirati funkcijski vmesnik, lahko definicijo implementacijskega razreda in njegove edine abstraktne metode nadomestimo s t.i. lambdo
- splošna oblika lambde

```
(Tip1 param1, Tip2 param2, ...) -> {
    telo metode
}
```

# Metoda primerjalnikPoPriimku na tri načine

- način 1: statični notranji razred

```
public static Comparator<Oseba> primerjalnikPoPriimku() {
    return new PrimerjalnikPoPriimku();
}

private static class PrimerjalnikPoPriimku
        implements Comparator<Oseba> {

    @Override
    public int compare(Oseba prva, Oseba druga) {
        return prva.priimek.compareTo(druga.priimek);
    }
}
```

# Metoda primerjalnikPoPriimku na tri načine

- način 2: anonimni notranji razred

```
public static Comparator<Oseba> primerjalnikPoPriimku() {
    return new Comparator<Oseba>() {
        @Override
        public int compare(Oseba prva, Oseba druga) {
            return prva.priimek.compareTo(druga.priimek);
        }
    };
}
```

# Metoda primerjalnikPoPriimku na tri načine

- način 3: lambda

```
public static Comparator<Oseba> primerjalnikPoPriimku() {
    return (Oseba prva, Oseba druga) -> {
        return prva.priimek.compareTo(druga.priimek);
    };
}
```

# Lambda

- lambda združuje definicijo implementacijskega razreda in abstraktne metode
- lambda je izraz
- kot vsak izraz ima svojo vrednost in tip
- vrednost lambde je kazalec na objekt funkcijskega vmesnika
- tip lambde se določi iz konteksta

# Določitev tipa lambde

- metoda `primerjalnikPoPriimku` vrne vrednost tipa `Comparator<Oseba>`

```
public static Comparator<Oseba> primerjalnikPoPriimku() {
    return (Oseba prva, Oseba druga) -> {
        return prva.priimek.compareTo(druga.priimek);
    }
}
```

- to pomeni, da je tip lambde

```
(Oseba prva, Oseba druga) -> {
    return prva.priimek.compareTo(druga.priimek);
}
```

enak `Comparator<Oseba>`

# Uporaba lambde

- lambda je objekt funkcijskega vmesnika, zato jo uporabimo tako, da pokličemo metodo vmesnika

```
// koda v testnem razredu;
// do atributa priimek dostopamo z getterjem
Comparator<Oseba> poPriimku = (Oseba prva, Oseba druga) -> {
    return prva.vrniPriimek().compareTo(druga.vrniPriimek());
};
Oseba joze = new Oseba("Jože", "Gorišek", 'M', 1956);
Oseba janez = new Oseba("Janez", "Novak", 'Z', 1973);
System.out.println(poPriimku.compare(joze, janez));
```

# Poenostavitev št. 1

- če telo lambde vsebuje zgolj stavek `return` ...

```
(Tip1 param1, Tip2 param2, ...) -> {
    return izraz;
}
```

- ... potem lahko namesto `{ return izraz; }` pišemo samo *izraz*:

```
(Tip1 param1, Tip2 param2, ...) -> izraz
```

# Poenostavitev št. 1

- namesto

```
(Oseba prva, Oseba druga) -> {
    return prva.priimek.compareTo(druga.priimek);
}
```

- lahko pišemo

```
(Oseba prva, Oseba druga) ->
        prva.priimek.compareTo(druga.priimek)
```

# Poenostavitev št. 2

- tipe parametrov lahko pogosto izpustimo
- prevajalnik jih določi sam
- namesto

```
(Oseba prva, Oseba druga) ->
        prva.priimek.compareTo(druga.priimek)
```

lahko pišemo

```
(prva, druga) -> prva.priimek.compareTo(druga.priimek)
```

- ker je lambda tipa `Comparator<Oseba>`, implementira metodo `public int compare(Oseba prva, Oseba druga)`, zato prevajalnik sam ugotovi, da sta parametra `prva` in `druga` tipa `Oseba`

# Lambdi pri urejanju oseb

```
public static Comparator<Oseba> primerjalnikPoPriimku() {
    return (prva, druga) -> prva.priimek.compareTo(druga.priimek);
}

public static Comparator<Oseba> primerjalnikPoSpoluInStarosti() {
    return (prva, druga) -> {
        if (prva.spol != druga.spol) {
            return druga.spol - prva.spol;
        }
        return prva.letoRojstva - druga.letoRojstva;
    };
}
```

# Primer: tabela dvojiške operacije

- napišimo metodo `tabelaOperacije`, ki izpiše tabelo podane dvojiške operacije za števila od 1 do $n$
- pričakovani izpis za operacijo + in $n = 5$:

```
  2   3   4   5   6
  3   4   5   6   7
  4   5   6   7   8
  5   6   7   8   9
  6   7   8   9  10
```

# Predstavitev dvojiške operacije

- dvojiško operacijo predstavimo kot objekt vmesnika, ki vsebuje abstraktno metodo, ki sprejme dve celi števili in vrne celoštevilski rezultat
- lahko napišemo svoj vmesnik, lahko pa uporabimo vmesnik `IntBinaryOperator` iz paketa `java.util.function`

```
public interface IntBinaryOperator {
    public abstract int applyAsInt(int left, int right);
}
```

# Metoda tabelaOperacije

```
public static void tabelaOperacije(int n, IntBinaryOperator op) {
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            System.out.printf(" %3d", op.applyAsInt(i, j));
        }
        System.out.println();
    }
}
```

# Izdelava objekta tipa IntBinaryOperator

- s pomočjo statičnega notranjega razreda

```
public static void main(String[] args) {
    tabelaOperacije(10, new Plus());
    tabelaOperacije(10, new Minus());
}
private static class Plus implements IntBinaryOperator {
    @Override
    public int applyAsInt(int a, int b) {
        return a + b;
    }
}
private static class Minus implements IntBinaryOperator {
    @Override
    public int applyAsInt(int a, int b) {
        return a - b;
    }
}
```

# Izdelava objekta tipa IntBinaryOperator

- s pomočjo anonimnega notranjega razreda

```
public static void main(String[] args) {
    tabelaOperacije(10, new IntBinaryOperator() {
        @Override
        public int applyAsInt(int a, int b) {
            return a + b;
        }
    });

    tabelaOperacije(10, new IntBinaryOperator() {
        @Override
        public int applyAsInt(int a, int b) {
            return a - b;
        }
    });
    ...
}
```

# Izdelava objekta tipa `IntBinaryOperator`

- s pomočjo lambde

```
public static void main(String[] args) {
    tabelaOperacije(10, (a, b) -> a + b);
    tabelaOperacije(10, (a, b) -> a - b);
    ...
}
```

- ker metoda `tabelaOperacije` kot drugi parameter sprejme objekt tipa `IntBinaryOperator`, je tip lambde `IntBinaryOperator`
- metoda `applyAsInt`, ki jo lambda implementira, sprejme parametra tipa `int`, zato je to tudi tip parametrov `a` in `b` v lambdi

# Lokalne spremenljivke v lambdi

- v lambdi lahko uporabljamo lokalne spremenljivke in parametre oklepajoče metode, toda le tiste, ki se ne spreminjajo
- množilnik s podanim faktorjem

```
public static IntUnaryOperator mnozilnik(int faktor) {
    // spremenljivke faktor ne smemo spreminjati!
    return n -> n * faktor;
}

public static void main(String[] args) {
    IntUnaryOperator krat5 = mnozilnik(5);
    System.out.println(krat5.applyAsInt(3));   // 15
    System.out.println(krat5.applyAsInt(10));  // 50
}
```

# Zbirke in lambde

- lambde nam pogosto pridejo prav pri klicih metod, ki se sprehajajo po podani zbirki in sproti obdelujejo njene elemente
- tipični primeri
  - štetje elementov, ki izpolnjujejo podani pogoj
  - izvršitev določenega opravila za vsak element zbirke
  - združevanje elementov z dvojiškim operatorjem
  - grupiranje elementov po rezultatih funkcije
  - ...

# Štetje elementov, ki izpolnjujejo pogoj

- `public static <T> int prestej(`
  `Collection<T> zbirka,` *`pogoj`*`)`
- pogoj predstavimo kot objekt funkcijskega vmesnika, ki vsebuje abstraktno metodo, ki sprejme objekt tipa `T` in vrne `boolean`
- tak vmesnik je

```
public interface Predicate<T> {
    public abstract boolean test(T t);
}
```

- pri klicu metode `prestej` lahko objekt vmesnika `Predicate` izdelamo kot lambdo (z njo implementiramo metodo `test`)

# Štetje elementov, ki izpolnjujejo pogoj

```
public static <T> int prestej(Collection<T> zbirka, Predicate<T> pogoj) {
    int stevec = 0;
    for (T element: zbirka) {
        if (pogoj.test(element)) {
            stevec++;
        }
    }
    return stevec;
}

public static void main(String[] args) {
    List<Integer> stevila = List.of(20, 15, 32, 7, 19, 14, 23, 35);
    int stSodih = prestej(stevila, n -> n % 2 == 0);   // 3

    List<String> imena = List.of("Ana", "Branko", "Cvetka", "Denis");
    int stImenDolzine5 = prestej(imena, ime -> ime.length() == 5);  // 1
}
```

# Izvedba opravila za vsak element zbirke

- `public static <T> void zaVsak(`
  `Collection<T> zbirka,` *opravilo*`)`
- opravilo predstavimo kot objekt vmesnika, ki vsebuje metodo, ki sprejme objekt tipa `T` in ne vrne ničesar
- tak vmesnik je

```
public interface Consumer<T> {
    public abstract void accept(T t);
}
```

# Izvedba opravila za vsak element zbirke

```
public static <T> void zaVsak(Collection<T> zbirka,
                              Consumer<T> opravilo) {
    for (T element: zbirka) {
        opravilo.accept(element);
    }
}

public static void main(String[] args) {
    List<Integer> stevila = List.of(20, 15, 32, 7, 19, 14, 23, 35);
    zaVsak(stevila, n -> {
        System.out.println(n);
    });
    // vsak element seznama se izpiše v svojo vrstico

    List<String> imena = List.of("Ana", "Branko", "Cvetka", "Denis");
    Map<String, Integer> ime2dolzina = new TreeMap<>();
    zaVsak(imena, ime -> {
        ime2dolzina.put(ime, ime.length());
    });
    // ime2dolzina: Ana -> 3, Branko -> 6, Cvetka -> 6, Denis -> 5
}
```

# Združevanje elementov z dvojiškim operatorjem

- `public static <T> T zdruzi(Collection<T> zbirka, BinaryOperator<T> operator, T zacetek)`
- `zbirka`: zbirka z elementi $e_0, e_1, \ldots, e_{n-1}$ tipa `T`
- `operator`: objekt z metodo `T apply(T a, T b)`, ki predstavlja dvojiški operator (označimo ga s $\circ$)
- metoda vrne rezultat izraza

$$((((\texttt{zacetek} \circ e_0) \circ e_1) \circ e_2) \circ \ldots) \circ e_{n-1}$$

# Združevanje elementov z dvojiškim operatorjem

```
public static <T> T zdruzi(Collection<T> zbirka,
        BinaryOperator<T> operator, T zacetek) {

    T rezultat = zacetek;
    for (T element: zbirka) {
        rezultat = operator.apply(rezultat, element);
    }
    return rezultat;
}

public static void main(String[] args) {
    List<Integer> stevila = List.of(20, 15, 32, 7, 19, 14, 23, 35);
    int vsota = zdruzi(stevila, (a, b) -> a + b, 0);  // 165

    List<String> imena = List.of("Ana", "Branko", "Cvetka", "Denis");
    String najdaljseIme = zdruzi(
            imena, (a, b) -> (a.length() >= b.length() ? a : b), "");
            // Branko
}
```

# Grupiranje elementov po rezultatih funkcije

- `public static <T, R> Map<R, List<T>> grupiraj(`
  `Collection<T> zbirka, Function<T, R> funkcija)`
- `funkcija`: objekt z metodo `R apply(T a)`
- metoda uporabi funkcijo nad vsakim elementom zbirke in izdela slovar, ki vrednost *r* preslika v seznam elementov zbirke, pri katerih je rezultat funkcije enak *r*

# Grupiranje elementov po rezultatih funkcije

```
public static <T, R> Map<R, List<T>> grupiraj(
        Collection<T> zbirka, Function<T, R> funkcija) {

    Map<R, List<T>> slovar = new HashMap<>();
    for (T element: zbirka) {
        R rezultat = funkcija.apply(element);
        List<T> elementiZaRezultat = slovar.get(rezultat);
        if (elementiZaRezultat == null) {
            elementiZaRezultat = new ArrayList<T>();
            slovar.put(rezultat, elementiZaRezultat);
        }
        elementiZaRezultat.add(element);
    }
    return slovar;
}
```

# Grupiranje elementov po rezultatih funkcije

```
public static void main(String[] args) {
    List<Integer> stevila = List.of(
            20, 15, 32, 7, 19, 14, 23, 35);
    Map<Boolean, List<Integer>> sodost2stevila =
            grupiraj(stevila, n -> n % 2 == 0);
            // false -> [15, 7, 19, 23, 35], true -> [20, 32, 14]

    List<String> imena = List.of(
            "Ana", "Branko", "Cvetka", "Denis");
    Map<Integer, List<String>> dolzina2imena =
            grupiraj(imena, ime -> ime.length());
            // 3 -> [Ana], 5 -> [Denis], 6 -> [Branko, Cvetka]
}
```