Objektno orijentirano programiranje

11. Ugniježđene i anonimne klase. Lambda izrazi.

Ugniježđene klase

- statičko i nestatičko gniježđenje

```
public class OuterClass {
   static class StaticNestedClass {        …      }
   class InnerClass {        …      }
}
```

- klasa *StaticNestedClass* definirana je kao statička
  - instanca ove klase može postojati bez postojanja instance vanjske klase (ne postoji povezanost)
- klasa *InnerClass* nije definiran uporabom ključne riječi *static*
  - ugniježđene klase koje nisu definirane kao statičke nazivaju se unutarnje klase (engl. Inner classes).
  - instanca unutarnje klase ne može postojati bez instance vanjske klase (povezana je s njom)

Primjer za demonstraciju ugniježđene i unutarnje klase (1)

- Klasa *FixedSizeCollection<T>* za kolekciju fiksne veličine.
- Želimo implementirati sučelje *Iterable<T>* da se kolekcija može koristiti u kratkom obliku for petlji, npr.

```
FixedSizeCollection<String> col = new
   FixedSizeCollection<>(
                     "Ana", "Ivana", "Jasmina");
for(String s1 : col)
       for(String s2 : col)
              System.out.println(s1+", "+s2);
```

```
Ana, Ana
Ana, Ivana
Ana, Jasmina
Ivana, Ana
Ivana, Ivana
Ivana, Jasmina
Jasmina, Ana
Jasmina, Ivana
Jasmina, Jasmina
```

Primjer za demonstraciju ugniježđene i unutarnje klase (2)

- Sučelje `Iterable<T>` definira samo jednu metodu

```
public Iterator<T> iterator()
```

- Sučelje `Iterator<T>` definira dvije metode

```
public boolean hasNext()
public T next()
```

- Nad kolekcijom se istovremeno može obavljati više iteracija, pa svaki primjerak iteratora mora pamtiti svoje vlastito stanje.
  - rezultat poziva mora biti novi primjerak iteratora.
  - rješenje bez unutarnjih ili ugniježđenih klasa:
    - Što kad kolekcija ne bi nudila mogućnost dohvata po indeksu?
- Iterator ćemo prvo izvesti korištenjem ugniježđene statičke klase, a zatim korištenjem unutarnje klase.

`hr.fer.oop.iterable.*`

Statička ugniježđena klasa (1)

- Obično se koriste za definiranje pomoćne strukture podataka.
  - za iteratore je uobičajena unutarnja klasa
- Ako je ugniježđena klasa javna, može se direktno instancirati s `new OuterClass.InnerClass(…`
- Želimo li osigurati da ugniježđena statička klasa može dohvatiti članske varijable vanjske klase, eksplicitno kroz konstruktor ugniježđene klase mora poslati referencu na vanjsku klasu te se takva referenca čuva u ugniježđenoj klasi.

```
public class FixedSizeCollection<T> implements Iterable<T>
   {
   ...
   @Override
   public Iterator<T> iterator() {
       return new CollectionIterator<>(this);
   }
   ...
```

hr.fer.oop.nestedstatic.*

Statička ugniježđena klasa (2)

- Primijetiti da ugniježđena klasa ima pravo pristupa privatnim varijablama vanjske klase!

```
public class FixedSizeCollection<T> implements Iterable<T> {
   ...
   private static class CollectionIterator<T> implements
   Iterator<T> {
       private FixedSizeCollection<T> col;
       private int index;
       private int size;
       public CollectionIterator(FixedSizeCollection<T> col){
              this.col = col; index = 0;
              this.size = col.elements.length;
       }
       @Override
       public boolean hasNext() {
              return index < size;
       } ...
```

Statička ugniježđena klasa (3)

```
public class FixedSizeCollection<T> implements Iterable<T> {
   ...
   private static class CollectionIterator<T> implements
   Iterator<T> {
       ...
       @Override
       public T next() {
              if(!hasNext()) {
                     throw new NoSuchElementException(
                            "No more elements are
   available.");
              }
              return col.elements[index++];
       }
   }
```

Unutarnje klase

- Ugniježđena klasa koja nije definirana kao statička naziva se **unutarnja klasa**.
- Primjerci te klase moraju se stvarati isključivo u kontekstu primjerka vanjske klase kako bi mogli pokupiti referencu `this` na primjerak vanjske klase koja ih stvara.
  - Npr.
```
OuterClass outer = new OuterClass();
InnerClass inner = outer.new InnerClass();
```
- Posljedica je da primjerci unutarnje klase imaju pristup članskim varijablama i metodama primjerka vanjske klase koja ih je stvorila.
- Unutarnja klasa *InnerClass* u bilo kojoj metodi može napisati `OuterClass.this` i to predstavlja referencu na primjerak vanjske klase.

Unutarnja klasa za implementaciju iteratora

```
public class FixedSizeCollection<T> implements Iterable<T> {
   ...
   private class CollectionIterator implements Iterator<T>{
       private int index = 0;
       @Override
       public boolean hasNext() {
         return index < elements.length;
       }

       @Override
       public T next() {
         if(!hasNext()) {
           throw new NoSuchElementException(
                   "No more elements are available.");
         }
         return elements[index++];
       } ...

   }
```

hr.fer.oop.inner.*

Instanciranje unutarnje klase iz vanjske klase

```
public class FixedSizeCollection<T> implements Iterable<T> {
   ...
   @Override
   public Iterator<T> iterator() {
       return new CollectionIterator();
   }
```

hr.fer.oop.inner.*

- Članska metoda klase koja definira kolekciju, i u toj metodi implicitno je dostupna referenca *this* na primjerak kolekcije
  - Nema potrebe za članskom varijablom tipa *FixedSizeCollection* u unutarnjoj klasi
    - Posljedično, unutarnja klasa nije trebala biti parametrizirana
- U metodi hasNext pitamo se `index < elements.length`
  - `=> this.index < FixedSizeCollection.this.elements.length;`

Još jedan primjer korištenja iteratora

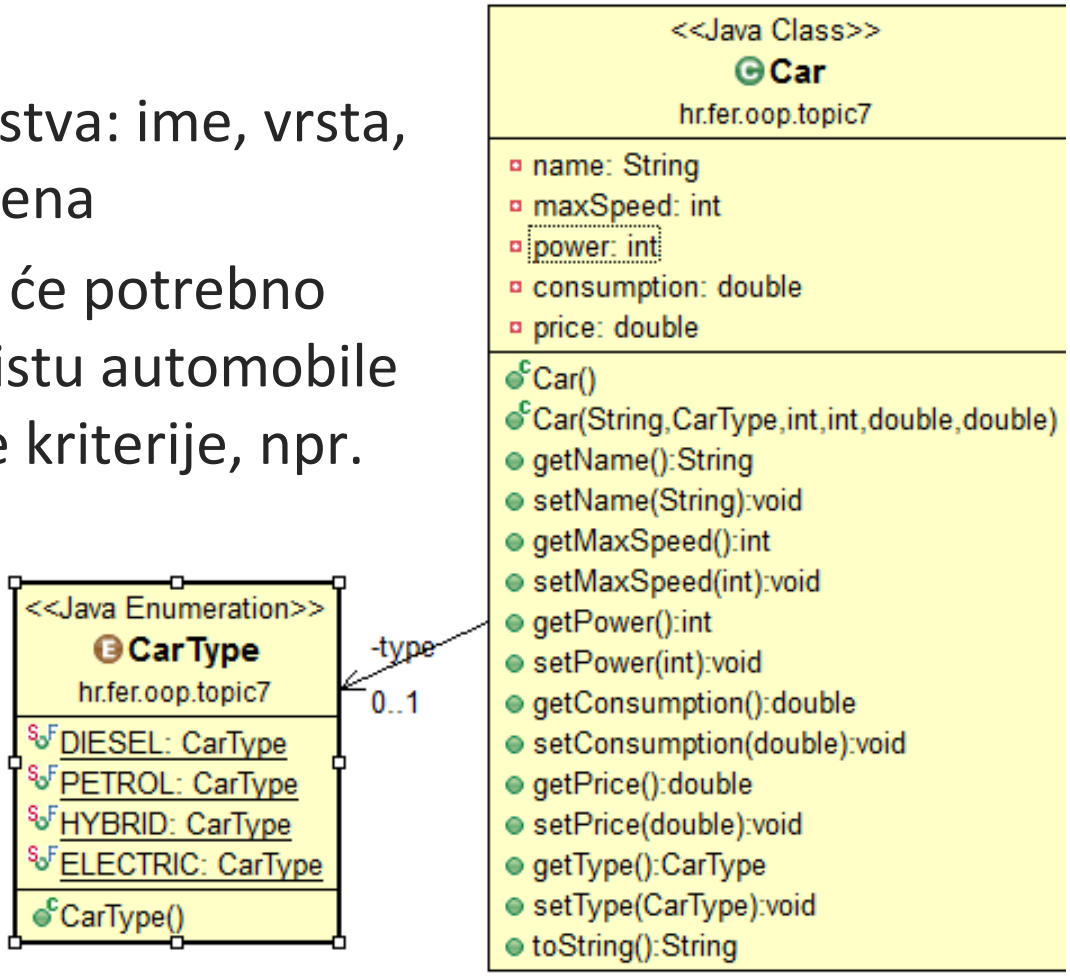
```
public static void main(String[] args){
   ...
   FixedSizeCollection<String> col1 =
                       new FixedSizeCollection<>("Pero", "Ivo");
   FixedSizeCollection<String> col2 =
                       new FixedSizeCollection<>("Ana", "Jasna");
   Iterator<String> it1 = col1.iterator();
   System.out.println(it1.next()); // Pero
   Iterator<String> it2 = col2.iterator();
   System.out.println(it2.next()); // Ana
   ...
```

hr.fer.oop.[inner,nestedstatic].Main.java

- Svaki od iteratora iterira po svojoj kolekciji.

Opis primjera

- U nekoj listi bit će pohranjeni auti različitih karakteristika.
- Svaki auto ima sljedeća svojstva: ime, vrsta, brzina, snaga, potrošnja i cijena
- U primjerima koji slijede bit će potrebno ispisati ili pohraniti u novu listu automobile koji zadovoljavaju određene kriterije, npr.
  - svi benzinski,
  - svi jači od 100KS,
  - svi dieseli jeftiniji od 100 000
- Lista vozila tvrdo kodirana u klasi *CarCatalog*



Primjer 1:

- Ispisati sve aute koji voze na diesel
  - U metodi za ispis koristimo samo funkcionalnost šetnje po nekom skupu podataka, pa je dovoljno da je taj skup podataka implementirao *Iterable* (ne mora nužno biti *List* ili neka druga kolekcija)

```
public class Main {
   public static void main(String[] args) {
        List<Car> cars = CarCatalog.loadCars();
        printDieselCars(cars);
   }
   private static void printDieselCars(Iterable<Car> cars) {
        for(Car car : cars)
               if (car.getType() == CarType.DIESEL){
                     System.out.println(car);
   }
```

hr.fer.oop.lambda.example1

- Što ako želimo ispisati sve benzince?
  - Nova metoda? Preimenovati metodu u printCars i dodati parametar carType?

Parametrizacija ponašanja

- Što da smo htjeli ispisati aute jeftinije od 100 000Kn?
  - Nova metoda ili proširiti prethodnu s još jednim parametrom i zastavicom koji od kriterija se koristi?
- Što s ostalim kriterijima ili kombinacijom kriterija?
- Primijetiti da bi većina koda ostala ista (šetnja, ispis) te da je razlika samo u izrazu ispitivanja uvjeta
  - Potrebno osmisliti način kako parametrizirati taj dio, a da ostatak ostane isti
    - *Slično postoji i u C-u, npr. quick sort: funkcija qsort prima pokazivač na polje kao void*, broje elemenata u polju, veličinu pojedinog elementa te pokazivač na funkciju koja uspoređuje dva elementa*

Predikat i funkcijska sučelja

- **Predikat** – metoda koja za neki objekt provjerava zadovoljava li neki uvjet (vraća istinu ili laž)
- Java ima sučelje *Predicate<T>* s metodom

```
boolean test(T t)
```

- Metoda test je jedina metoda koju neka klasa treba napisati prilikom implementacije sučelja *Predicate*
- Sučelja sa samo jednom (apstraktnom) metodom (tj. samo jednom metodom koju treba implementirati) nazivaju se **funkcijska sučelja** (engl. functional interface)
  - Činjenica koja se koristi kod lambda izraza
  - Može se označiti s @FunctionalInterface – oznaka prevoditelju

Primjer predikata za automobil

Datoteka CheapCarPredicate.java

```
package hr.fer.oop.lambda.example2;
import java.util.function.Predicate;
import hr.fer.oop.lambda.Car;
public class CheapCarPredicate implements Predicate<Car> {
        @Override
        public boolean test(Car car) {
                return car.getPrice() < 100000;
        }
}
```

Datoteka DieselCarPredicate.java

```
package hr.fer.oop.lambda.example2;
import java.util.function.Predicate;
import hr.fer.oop.lambda.Car;
public class DieselCarPredicate implements Predicate<Car> {
        @Override
        public boolean test(Car car) {
                return car.getType() == CarType.DIESEL;
        }
}
```

Korištenje predikata u metodi za ispis automobila

- Metoda za ispis auta promijenjena je tako da prima konkretni predikat kojim se u if naredbi testira da li je auto kandidat za ispis

```
public class Main {
   public static void main(String[] args) {
        ArrayList<Car> cars = CarCatalog.loadCars();
        printCars(cars, new CheapCarPredicate());
        printCars(cars, new DieselCarPredicate());
   }

   private static void printCars(
                        Iterable<Car> cars, Predicate<Car> predicate)
   {
        for(Car car : cars)
               if (predicate.test(car))
                        System.out.println(car);
   }
```

hr.fer.oop.lambda.example2

Efekt uvođenja predikata u metodu za ispis auta

- Uočite što smo ovime postigli
- Uklonili smo dupliciranje koda
  - više nemamo dvije metode koje sadrže kompletnu logiku obrade auta (do na razlike u if-u)
- Imamo jednu metodu koja prima predikat
  - dodavanje novih kriterija odabira auta za ispis ne zahtjeva izmjenu metode *printCars*
  - naš kôd npr. možemo izvesti u *jar*, a korisnik ga može koristiti po želji sam pišući vlastite predikate na način da napiše novu klasu koja implementira sučelje *Predicate<Car>*

Anonimne klase

- U prethodnom slučaju predikati su bili napisani u zasebnim klasama i koristili smo ih samo jednom (i vjerojatno bez namjere da nam trebaju kasnije)
- Java nudi mogućnost da za takve slučajeve definiramo anonimnu klasu (bez imena) za koju se na mjestu definiranja odmah i stvori primjerak te klase

hr.fer.oop.lambda.example3

```
public static void main(String[] args) {
    ArrayList<Car> cars = CarCatalog.loadCars();
    printCars(cars, new Predicate<Car>(){
        @Override
        public boolean test(Car car) { return car.getPrice() < 100000; }
    });
    printCars(cars, new Predicate<Car>(){
        @Override
        public boolean test(Car car) { return car.getType() ==
          CarType.DIESEL; }
    });
}
```

Definiranje anonimne klase

- Uočite da je *Predicate* sučelje, pa je `new Predicate<Car>()` samo po sebi besmislica, ali u ovom slučaju taj redak ne znači da stvaramo primjerak sučelja
- Interpretacija je: stvaramo primjerak anonimne klase koji implementira sučelje *Predicate<Car>* pri čemu je implementacija navedena u vitičastim zagradama u nastavku

```
printCars(cars, new Predicate<Car>() {
        @Override
        public boolean test(Car car) {
            return car.getType() == CarType.DIESEL;
        }
    });
}
```

hr.fer.oop.lambda.example3

Svojstva anonimnih klasa

- Anonimne klase mogu se definirati na temelju sučelja ili na temelju druge klase (obične ili apstraktne)
  - Naravno, da bi se mogao stvoriti primjerak klase, definicija anonimne klase mora “pokrpati” sve razloge zbog kojih je sučelje ili bazna klasa apstraktna: anonimna klasa ne smije biti apstraktna
- S obzirom da anonimna klasa nema imena, ne može imati eksplicitan konstruktor, ali može imati inicijalizacijski blok
- Po svemu ostalome, ponaša se slično kao lokalna (unutarnja) klasa – klasa koja je definirana u tijelu metode
  - Lokalne klase se rijetkom upotrebljavaju
  - Metode takve klase u Javi 8 imaju pristup lokalnim varijablama i parametrima metode u kojoj su definirane ako su one **finalne ili efektivno-finalne**
    - Efektivno-finalna varijabla je ona varijabla koja se nakon inicijalizacije više ne mijenja
  - Proučiti: http://docs.oracle.com/javase/tutorial/java/javaOO/localclasses.html

Anonimne klase i referenca *this*

- Anonimne klase definiraju novi doseg (engl. scope)
- U metodama anonimne klase referenca `this` se odnosi na primjerak anonimne klase – ne na primjerak vanjske klase
- Anonimna klasa stvorena u nestatičkom kontekstu vanjskom objektu može pristupiti sintaksom `ImeKlase.this`
- Anonimna klasa može svojim metodama sakriti varijable iz vanjske metode/klase ako definira svoje varijable istoga imena (engl. *shadowing*)

Lambda izrazi umjesto anonimne klase

- Pisanje kompletne deklaracije anonimne klase na mjestu gdje je potrebna traži mnoštvo kôda koji dosta otežava čitljivost napisanoga
- U slučaju da sučelje deklarira samo jednu metodu (funkcijsko sučelje), od Jave 8 moguće je čitavu deklaraciju i stvaranje primjerka anonimne klase opisati sažetom sintaksom koja sadrži konkretne naredbe

hr.fer.oop.lambda.example4

```
printCars(cars, (Car car) -> {
                return car.getPrice() < 100000);
        }
);
printCars(cars, (Car car) -> car.getType() == CarType.DIESEL);
```

Lambda izrazi

- Kao drugi argument metodi *printCars* predan je odsječak koda

```
(Car car) -> car.getType() == CarType.DIESEL
```

  koji označava da se radi o metodi koja kad bude pozvana s argumentom *car* tipa *Car* kao rezultat vraća izraz desno od strelice
- Prevoditelj iz deklaracije metode `printCars` pronalazi da drugi argument mora biti referenca na primjerak klase koja implementira sučelje *Predicate<Car>*
- Potom pronalazi da to sučelje ima samo jednu metodu (`test`) koja prima jedan argument tipa *Car* i vraća vrijednost tipa *boolean*
  - Stoga, može i `(car) -> car.getType() == CarType.DIESEL`
- Ako iza strelice stavimo vitičaste zagrade (ako npr. želimo napisati više naredbi), a očekuje se da lambda izraz vraća vrijednost, onda je potrebno napisati *return* za vraćanje vrijednosti iz tijela lambda izraza

```
printCars(cars, (car) -> {return car.getPrice() < 100000;});
```

Lambda izrazi i *this*

- Iako se lambda izrazi ponašaju slično kao anonimne klase, budući da generiranje tog kôda obavlja prevoditelj, pravila su nešto drugačija, nego kada sami pišemo kôd anonimne klase
- Lambde ne uvode novi doseg
- Stoga se *this* u lambdi odnosi na isto što i *this* izvan lambde što nije bio slučaj kod anonimnih klasa!

Funkcijsko sučelje *Consumer<T>*

- Proširimo prethodni primjer tako da ispis auta koji zadovoljava određeni predikat izraz ne bude čvrsto kodiran i da se da po potrebi može promijeniti
- Ispis je akcija na jednom elementu tipa *Car* za koji treba nešto obaviti, što znači da je povratna vrijednost takve metode *void*
  - U Javi već postoji funkcijsko sučelje koje ima takvu metodu: `Consumer<T>` s metodom `void accept(T t)`

```
private static void printCars(Iterable<Car> cars,
                Predicate<Car> predicate,
                Consumer<Car> action) {
  for(Car car : cars)
      if (predicate.test(car)) action.accept(car);
}
```

hr.fer.oop.lambda.example5

- Primjer poziva metode:

```
printCars(cars, (car) -> car.getPrice() < 100000,
        (car) -> System.out.println("Cheap car: " + car));
```

Koju vrstu klasa kada koristiti?

- Lambda izraz
  - za definiranje „jednostavnog” ponašanja (npr. opis što napraviti sa svakim elementom iz skupa) koje treba proslijediti negdje drugdje u kodu
- Anonimna klasa
  - u slučajevima kad lambda izraz nije prikladan, jer treba definirati dodatne atribute ili metode
- Lokalna klasa
  - kad se klasa ne koristi nigdje van metode u kojoj je definirana, ali postoji više instanci klase, trebamo konstruktor ili je jednostavno potreban imenovani tip zbog dodatnih metoda ili varijabli
- Ugniježđene statičke klase
  - u slučajevima sličnim lokalnoj klasi, ali kad je potrebna šira vidljivost klase
- Unutarnja klasa:
  - kad je potrebno pristupati privatnim članovima vanjske klase
- https://docs.oracle.com/javase/tutorial/java/javaOO/whentouse.html

Još jedan primjer korištenja lambda izraza

- Napisati metodu koja će u listi auta pronaći dva najsličnija auta i za njih napraviti određenu akciju.
- U trenutku pisanja koda akcija koju treba izvršiti nije poznata
  - npr. to može biti ispis na ekran, ali i smanjenje razlike u cijeni
- Kriterij za utvrđivanje sličnosti auta nije poznat prilikom pisanja metode, ali zna se da je to funkcija koja za dva automobila vraća neki cijeli broj
- Rješenje:
  - Akciju i funkciju sličnosti modelirati posebnim sučeljima
  - Za demonstraciju metode koristiti lambda izraze

Potrebna sučelja za rješenje primjera

- Akcija na najsličnijem paru auta znači da će postojati metoda koja prima dva argumenta tipa *Car* i za njih nešto obavi, što znači da je povratna vrijednost takve metode *void*
  - U Javi već postoji funkcijsko sučelje koje ima takvu metodu: `BiConsumer<T, U>` s metodom `void accept(T t, U u)`
- Metoda za usporedbu prima dva auta i vraća cijeli broj. Potrebno je sučelje s metodom tipa *int* i dva argumenta tipa *Car*
  - U Javi već postoji funkcijsko sučelje koje možemo iskoristit: `BiFunction<T, U, R>` s metodom `R apply(T t, U u)`
- Naša metoda stoga ima deklaraciju
  - ```
    void theMostSimilarCar(Iterable<Car> cars,
     BiFunction<Car, Car, Integer> similarity,
     BiConsumer<Car, Car> action)
    ```
    ```

Lokalna klasa za pohranu para najsličnijih auta

- Par auta je potreban samo unutar ove metode, pa koristimo lokalnu klasu

```
public static void theMostSimilarCar(Iterable<Car> cars,
        BiFunction<Car, Car, Integer> similarity,
        BiConsumer<Car, Car> action) {

                class CarPair {
                        public Car first, second;
                        public CarPair(Car first, Car second) {
                                this.first = first;
                                this.second = second;
                        }
                }

                CarPair pair = null;
                ...
```

hr.fer.oop.lambda.example6

Ostatak metode za par najsličnijih auta

```
public static void theMostSimilarCar(Iterable<Car> cars,
        BiFunction<Car, Car, Integer> similarity,
        BiConsumer<Car, Car> action) {
        ...
  int min = Integer.MAX_VALUE;
  for(Car first : cars) {
    for(Car second : cars) {
      if (first == second) continue;
      if (pair == null
          || similarity.apply(first, second) < min) {
        pair = new CarPair(first, second);
        min = similarity.apply(first, second);
      }
    }
  }
  if (pair != null) action.accept(pair.first,  pair.second);
```

hr.fer.oop.lambda.example6

Primjeri poziva metode za par najsličnijih auta

- Predamo dva lambda izraza za argumente metode za 2 najsličnija auta

hr.fer.oop.lambda.example6

```
public static void main(String[] args) {
  ArrayList<Car> cars = CarCatalog.loadCars();
  theMostSimilarCar(cars,
       (a, b) -> (int) Math.abs(a.getPrice() - b.getPrice()),
       (a, b) -> System.out.format(
              "The most similar are: %n\t%s%n\t%s%n", a, b)
  );
}

public static void theMostSimilarCar(Iterable<Car> cars,
       BiFunction<Car, Car, Integer> similarity,
       BiConsumer<Car, Car> action){...
```

Referenca na postojeću metodu umjesto lambda izraza

- Ako već postoji gotova funkcija koja po svom potpisu odgovara metodi funkcijskog sučelja može se predati referenca na tu metodu.

hr.fer.oop.lambda.example7

```
public class Main {
  public static void main(String[] args) {
    ArrayList<Car> cars = CarCatalog.loadCars();
    theMostSimilarCar(cars,
         Main::distance),
         (a, b) -> System.out.format(
              "The most similar are: %n\t%s%n\t%s%n", a, b)
    );
  }
  private static int distance(Car a, Car b) { ... }
  public static void theMostSimilarCar(Iterable<Car> cars,
       BiFunction<Car, Car, Integer> similarity,
       BiConsumer<Car, Car> action){...
```