

# Objektno orijentirano programiranje

---

## **11. Ugniježdene i anonimne klase. Lambda izrazi.**

# Creative Commons

You are free to

- **Share** — copy and redistribute the material in any medium or format
- **Adapt** — remix, transform, and build upon the material

under the following terms

- **Attribution** — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
- **NonCommercial** — You may not use the material for commercial purposes.
- **ShareAlike** — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.
- <https://creativecommons.org/licenses/by-nc-sa/4.0/>



## *Iterator i Iterable (1/2)*

- Ako klasa implementira *Iterable*, može se koristiti unutar *for each* petlje
  - npr. *Collection* extends *Iterable*
- Pretpostavimo da želimo definirati vlastitu klasu koja čuva broj i želimo iterirati po znamenkama tog broja
- Možemo izračunati znamenke, spremiti ih u (nepromjenjivu) listu i vratiti takvu listu kao rezultat.

```
MyNumber num = new MyNumber(12345);  
for(Integer digit : num.digits())  
    System.out.println(digit); // ispisuje 1 2 3 4 5
```

## Iterator i Iterable (2/2)

- Što ako želimo napisati sljedeći kod:

```
MyNumber num = new MyNumber(12345);  
for(Integer digit : num)  
    System.out.println(digit); // prints 1 2 3 4 5 respectively
```

- U tom slučaju *MyNumber* mora implementirati `Iterable<Integer>`
  - Sučelje `Iterable<T>` definira samo jednu apstraktnu metodu  
*public Iterator<T> iterator()*
  - Sučelje `Iterator<T>` definira dvije metode koje treba implementirati  
*public boolean hasNext()* i *public T next()*
- U nastavku prvo slijedi loša implementacija i diskusija što je loše, nakon čega slijedi ispravno rješenje i poboljšanje točnog rješenja korištenjem ugniježđenih i unutarnjih klasa.

# Početni pokušaj – pohrana znamenki u nešto što je iterabilno

```
public class MyNumber implements Iterable<Integer> {  
    private int num;  
    public MyNumber(int num) { this.num = num; }  
    @Override  
    public Iterator<Integer> iterator() {  
        List<Integer> list = new LinkedList<>();  
        int temp = num;  
        while(temp > 0) { //pretpostavka: num je pozitivan  
            list.add(0, temp % 10);  
            temp /= 10;  
        }  
        return list.iterator();  
    }  
    ...  
}
```

11\_InnerNestedLambda/hr/fer/oop/iterable/wrong/MyNumber.java

- Što je loše u ovom rješenju (osim činjenice da nije prikazano kako implementirati sučelje *Iterator*)?
  - Podaci se nepotrebno pripremaju unaprijed (npr. možda će pozivatelj prestati s iteriranjem nakon prve pojave neparne znamenke) i zauzima memoriju (npr. kad bi veliki brojevi bili prikazani kao stringovi)

# Novi iterator → nova klasa

- Svaki put kad netko želi iterirati po znamenkama broja, treba nastati nova instanca iteratora
  - Svaki iterator pamti svoju „poziciju”

```
package hr.fer.oop.iterable;
import java.util.Iterator;
public class MyNumber implements Iterable<Integer> {
    private int num;
    public MyNumber(int num) {
        this.num = num;
    }

    @Override
    public Iterator<Integer> iterator() {
        return new DigitIterator(num);
    }
}
```

11\_InnerNestedLambda/hr/fer/oop/iterable/MyNumber.java

# Kako dobiti znamenke u željenom poretku (1)

- Broj znamenki je  $\lfloor \log n + 1 \rfloor$  pa se koriste operacije dijeljenja i ostatka dijeljenja s potencijama broja 10.
- Sljedeća znamenka postoji ako je broj pozitivan
  - U svakom koraku „nestane” jedna znamenka dok broj ne postane 0

```
public class DigitIterator implements Iterator<Integer> {
    private int expOf10;
    private int num;
    public DigitIterator(int num) {
        this.num = num;
        expOf10 = (int) Math.pow(10, (int) Math.log10(num));
    }
    @Override
    public boolean hasNext() {
        return num > 0;
    }
    ...
}
```

11\_InnerNestedLambda/hr/fer/oop/iterable/DigitIterator.java

## Kako dobiti znamenke u željenom poretku (2)

- Ako sljedeća znamenka postoji, dobije se kao ostatak pri dijeljenju s trenutnom potencijom broja 10 te se potencija pripremi za sljedeći korak

```
public class DigitIterator implements Iterator<Integer> {  
    ...  
    @Override  
    public Integer next() {  
        if (hasNext()) {  
            int digit = num / expOf10;  
            num %= expOf10;  
            expOf10 /= 10;  
            return digit;  
        }  
        else  
            throw new NoSuchElementException("No more digits");  
    }  
}
```

11\_InnerNestedLambda/hr/fer/oop/iterable/Digitlterator.java



# Trebamo li znati za *Digitliterator*?

- Jedino što moramo znati jest da je *MyNumber* implementirao *Iterable* i da vraća primjerak iteratora.
  - Nije potrebno znati da je taj iterator implementiran u klasi koja se zove *Digitliterator*
    - U većini slučajeva ta klasa nema svrhu izvan konteksta iteriranja, tj. nema svrhe van klase *MyNumber*

```
public class Main {  
    public static void main(String[] args) {  
        MyNumber number = new MyNumber(12345);  
        for(Integer digit : number)  
            System.out.println(digit); // ispisuje 1 2 3 4 5  
        System.out.println();  
        for(Integer digit1 : number)  
            for(Integer digit2 : number)  
                System.out.printf("%d - %d %n", digit1 , digit2);  
    }  
}
```

11\_InnerNestedLambda/hr/fer/oop/iterable/Main.java

# Ugniježdene klase

- Java omogućava da se klase definiraju unutar drugih klasa.
- Takve klase nazivaju se ugniježdene klase
- Predstavlja način grupiranja klasa koje se koristi samo na određenim mjestima
  - Ako klasa ima svrhu samo u kontekstu neke druge klase, onda je logično ugniježditi je unutar te druge i održavati ih zajedno
  - Kod male ugniježdene klase je tako blizu mjesta korištenja
- Ugniježdene klase povećavaju enkapsulaciju:
  - Ugniježdene klase imaju pristup privatnim varijablama klase u kojoj se nalaze
  - Dodatno, ugniježdene klase mogu (ali ne moraju) biti skriven od vanjskog svijeta

# Tipovi ugniježđenih klasa

- statičko i nestatičko gniježđenje

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

- ugniježdene klase mogu se označiti sa *static* (**static nested classes**).
  - Ponašajno isto kao i „normalne” klase, ali unutar druge klase zbog logike grupiranja i sl.
  - instanca ove klase može postojati bez postojanja instance vanjske klase (ne postoji povezanost)  
*OuterClass.StaticNestedClass var = new OuterClass.StaticNestedClass();*
- ugniježdene klase koje nisu definirane kao statičke nazivaju se **unutarnje klase** (engl. **inner classes**) te instanca unutarnje klase ne može postojati bez instance vanjske klase (povezana je s njom)

# Iterator kao statička ugniježđena klasa

- *Digitlterator* je implementiran kao ugniježđena statička klasa i označen s *private*
  - *Napomena: Digitlterator* može pristupiti privatnim članovima klase *MyNumber* ako ima referencu na objekt tipa *MyNumber*

```
public class MyNumber implements Iterable<Integer> {
    private int num;
    public MyNumber(int num) {
        this.num = num;
    }
    11_InnerNestedLambda/hr/fer/oop/iterable/nested//MyNumber.java

    @Override
    public Iterator<Integer> iterator() {
        return new DigitIterator(num);
    }
    private static class DigitIterator implements Iterator<Integer> {
        ...
    }
}
```

# Iterator kao unutarnja klasa

- Unutarnja klasa je vezana uz instancu vanjske klase preko koje nastaje te ima pristup članskim varijablama i metodama primjerka vanjske klase koja ih je stvorila
- Ako koristi varijable istog imena, referenca na varijablu iz vanjske klase može se dobiti s *OuterClassName.this.variable*

```
public class MyNumber implements Iterable<Integer> {  
    private int num;  
    ...  
    public Iterator<Integer> iterator() {  
        return new DigitIterator();  
    }  
    private class DigitIterator implements Iterator<Integer> {  
        private int num;  
        public DigitIterator() {  
            this.num = MyNumber.this.num;  
            ...  
        }  
    }  
}
```

11\_InnerNestedLambda/hr/fer/oop/iterable/inner/MyNumber.java

# Variable tipa unutarnje klase

- Ako unutarnja klasa nije označena kao privatna (može biti `private`, `public`, `protected`, ili `package private` dok vanjska klasa može biti samo `public` ili `package private`), tada se može stvoriti objekt tipa unutarnje klase, ali samo kao dio vanjske klase.
  - 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.
  - Posljedično ne mogu imati statičke članove

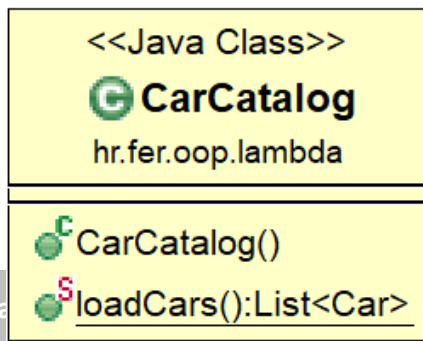
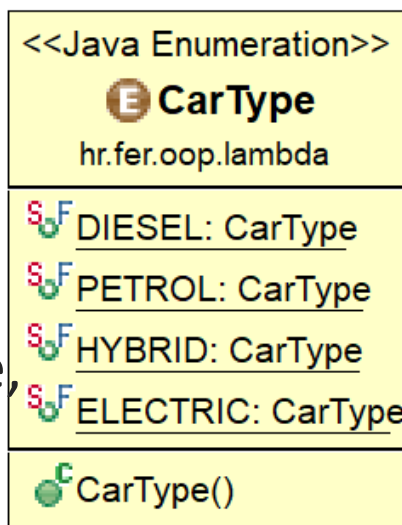
```
OuterClass outer = new OuterClass();  
InnerClass inner = outer.new InnerClass();
```

```
StaticNestedClass nested = new OuterClass.StaticNestedClass();
```

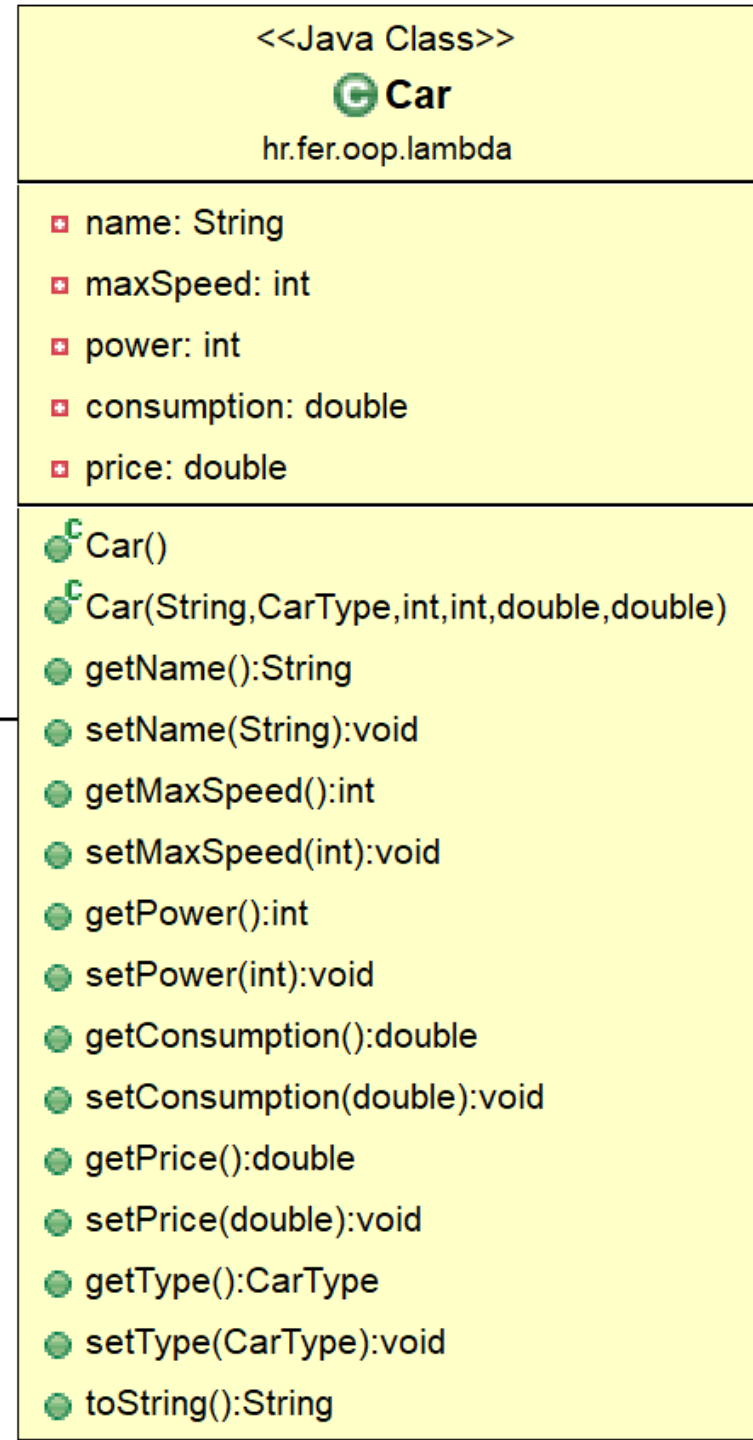
# Anonimne klase i lambda izrazi

# Opis primjera

- Auto ima sljedeća svojstva: ime, vrsta, brzina, snaga, potrošnja i cijena
- Lista vozila tvrdo kodirana u klasi *CarCatalog*
- U primjerima se ispisuju auti koji zadovoljavaju određene kriterije, npr.
  - svi benzinski
  - svi jači od 100KS,
  - svi dieseli jeftiniji od 100 000 ...



-type  
0..1





# Primjer #1 – ispisati sve aute koji voze na dizel

- U metodi se šetamo po kolekciji, pa je dovoljno da je argument Iterable (ne mora nužno biti List, može biti i neka druga kolekcija)

```
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);  
        }  
    }  
}
```

11\_InnerNestedLambda/hr/fer/oop/lambda/example1/Main.java

- 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 100 000 kn?
  - 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)
  - Označena s *@FunctionalInterface* (oznaka prevoditelju kojom detektira slučajno kršenje inicijalne namjene o jednoj metodi)
  - Napomena: Sučelje može imati *default* metode (vidi npr. *Predicate*)

## Primjer #2 – primjeri različitih predikata

- Inicijalno definirani u različitim klasama

11\_InnerNestedLambda/hr/fer/oop/lambda/example2/\*.java

```
public class CheapCarPredicate implements Predicate<Car> {  
    @Override  
    public boolean test(Car car) {  
        return car.getPrice() < 100000;  
    }  
}
```

```
public class DieselCarPredicate implements Predicate<Car> {  
    @Override  
    public boolean test(Car car) {  
        return car.getType() == CarType.DIESEL;  
    }  
}
```

## Primjer #2 – upotreba različitih predikata

- Metoda za ispis auta izmijenjena tako da prima predikat kojim se odlučuje hoće li se auto ispisati ili ne

```
public static void main(String[] args) {  
    List<Car> cars = CarCatalog.loadCars();  
    System.out.println("Cheap cars:");  
    printCars(cars, new CheapCarPredicate());  
    System.out.println("Diesel cars:");  
    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);  
    }  
}
```

11\_InnerNestedLambda/hr/fer/oop/lambda/example2/Main.java

# Efekt uvođenja predikata u metodu za ispis

- 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>*
    - Koncept koji se može vidjeti u nekoliko *default* metoda sučelja iz Java Collection Frameworka

## Primjer #3 – 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

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

11\_InnerNestedLambda/hr/fer/oop/lambda/example3/Main.java

# 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

11\_InnerNestedLambda/hr/fer/oop/lambda/example3/Main.java

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



# 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 od Java 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)
  - Anonimna klasa može svojim metodama sakriti varijable iz vanjske metode/klase ako definira svoje varijable istoga imena (engl. *shadowing*)
- U metodama anonimne klase referenca `this` se odnosi na primjerak anonimne klase, a ne na primjerak vanjske klase
  - Anonimna klasa stvorena u nestatičkom kontekstu vanjskom objektu može pristupiti sintaksom `ImeKlase.this`

## Primjer #4 – lambda izrazi

- 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), moguće je čitavu deklaraciju i stvaranje primjerka anonimne klase opisati sažetom sintaksom koja sadrži konkretne naredbe, tj. **lambda izrazom**

```
public static void main(String[] args) {  
    List<Car> cars = CarCatalog.loadCars();  
    printCars(cars, (car) -> {  
        return car.getPrice() < 100000;  
    });  
    printCars(cars, (car) -> car.getType() == CarType.DIESEL);  
}
```

11\_InnerNestedLambda/hr/fer/oop/lambda/example4/Main.java

# 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* zaključuje 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 idu vitičaste zagrade (ako u slučaju 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!

## Primjer #5 – 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);
        }
    }
}
```

11\_InnerNestedLambda/hr/fer/oop/lambda/example5/Main.java

## Primjer #5 – funkcijsko sučelje *Consumer<T>*

- Objekt tipa *Consumer<Car>* može se definirati na razne načine
- U primjeru učinjeno lambda izrazom

11\_InnerNestedLambda/hr/fer/oop/lambda/example5/Main.java

```
public static void main(String[] args) {
    List<Car> cars = CarCatalog.loadCars();
    printCars(cars,
        (car) -> car.getPrice() < 100000,
        (car) -> System.out.println("Cheap car: " + car)
    );
    printCars(cars,
        car -> car.getType() == CarType.DIESEL,
        car -> System.out.println("Diesel car: " + car)
    );
}

private static void printCars(Iterable<Car> cars,
    Predicate<Car> predicate, Consumer<Car> action) {
    ...
}
```

## Primjer #6 – *BiFunction* i *BiConsumer* (1/2)

- 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
- Sličnost je (matematički) funkcija  $f: Car \times Car \rightarrow \mathbb{Z}$ 
  - Funkcija dvije varijable tipa *Car* koja vraća cijeli broj
  - Java sadrži funkcijsko sučelje za ovu namjenu

```
public interface BiFunction<T, U, R>
```

s metodom *R*

```
    apply(T t, U u);
```

    - prima *t* (tipa *T*) i *u* (tipa *U*) i vraća rezultat tipa *R*
- *Consumer<T>* konzumira (izvršava akciju na objektu tipa) *T*
- *BiConsumer<T, U>* konzumira 2 objekta tipa *T* odnosno *U*



## Primjer #6 – *BiFunction* i *BiConsumer* (2/2)

- Potrebna sučelja implementirana su lambda izrazima

11\_InnerNestedLambda/hr/fer/oop/lambda/example6/Main.java

```
public static void main(String[] args) {
    List<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> distanceFunction,
    BiConsumer<Car, Car> action){
    ...
}
```

## Example #6 – Lokalne klase

- Par auta je potreban samo unutar ove metode, pa se može upotrijebiti lokalna klasa

```
public static void theMostSimilarCar(Iterable<Car> cars,
    BiFunction<Car, Car, Integer> distanceFunction,
    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;
    ...
}
```

11\_InnerNestedLambda/hr/fer/oop/lambda/example6/Main.java

## Primjer #6 – Ostatak koda

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

11\_InnerNestedLambda/hr/fer/oop/lambda/example6/Main.java

## Primjer #7 – Referenca na metodu

- Ako metoda odgovarajućeg potpisa već postoji, referenca na njuj se može dobiti s *Class::methodName* ili *object::methodName*
  - ovisi o tome je li metoda statička ili ne.

```
public static void main(String[] args) {
    List<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)
    );
}
11_InnerNestedLambda/hr/fer/oop/lambda/example7/Main.java

private static int distance(Car a, Car b) {
    return (int) Math.abs(a.getPrice() - b.getPrice());
}

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

# Koju vrstu klasa kada koristiti?

<https://docs.oracle.com/javase/tutorial/java/javaOO/whentouse.html>

- Lambda izraz
  - za „jednostavno” ponašanje (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 attribute 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
- Ugniježdene statičke klase
  - u slučajevima sličnim lokalnoj klasi, ali kad je potrebna šira vidljivost klase
- Unutarnja klasa:
  - kad je potrebno pristupati privatnim članovima vanjske klase

# Primjeri korištenja funkcijskih sučelja s *default* metoda sučelja *List* i *Map*

# Iterativni pristup ispisu sadržaja mape


- Neka se u *Map<CarType, Integer> carTypesCount* broje pojavljivanja svakog tipa vozila
- Iterativni pristup ispisu mape nalikuje sljedećem kodu

```
for(Map.Entry<CarType, Integer> entry : carTypesCount.entrySet()) {  
    System.out.format("%s occurred %d times. %n",  
        entry.getKey(), entry.getValue());  
}
```

- Od Java 8 u sučeljima *Map*, *List*, ... postoji nekoliko korisnih *default* metoda koje mogu zamijeniti ovaj pristup

# Metoda *forEach* u sučelju *Map*

- *Map* sadrži metodu *forEach* koja radi upravo ono što smo iterativno radili u prethodnom kodu, ali umjesto ispisa *forEach* izvršava neku akciju za svaki par na način da zove *void accept(K k, V v)* iz sučelja *BiConsumer <K, V>* te šalje ključ i vrijednosti
- Naš posao je napisati odgovarajući *BiConsumer*

```
java  Map.class ✕  
  
default void forEach(BiConsumer<? super K, ? super V> action) {  
    Objects.requireNonNull(action);  
    for (Map.Entry<K, V> entry : entrySet()) {  
        K k;  
        V v;  
        try {  
            k = entry.getKey();  
            v = entry.getValue();  
        } catch (IllegalStateException ise) {  
            // this usually means the entry is no longer in the  
            throw new ConcurrentModificationException(ise);  
        }  
        action.accept(k, v);  
    }  
}
```



# *forEach* u sučelju *Map* – primjer korištenja

- Korištenjem anonimne klase

```
Map<CarType, Integer> carTypesCount = new HashMap<>();  
...  
carTypesCount.forEach(new BiConsumer<CarType, Integer>() {  
    @Override  
    public void accept(CarType t, Integer u) {  
        System.out.println(t + " occurred " + u + " times");  
    }  
});  
11_InnerNestedLambda/hr/fer/oop/defmethods/ExampleMapCompute.java
```



- Korištenjem lamda izraza

```
Map<CarType, Integer> carTypesCount = new HashMap<>();  
...  
carTypesCount.forEach(  
    (type, num) ->  
        System.out.println(type + " occurred " + num + " times"));
```

11\_InnerNestedLambda/ hr/fer/oop/defmethods/ExampleMapComputeLambda.java

# Metoda *compute* u sučelju *Map*

- *compute* temeljem stare vrijednosti (null ako ključ nije bio u mapi) i zadanog argumenta, izračunava novu vrijednosti pridruženu ključu (ili uklanja ključ iz mape ako je izračunata vrijednosti null)
  - potreban objekt tipa *BiFunction*<K, V, V> za izračun
  - *compute* poziva *V apply(K k, V v)* šaljući ključ i staru vrijednost i koristi rezultat za odluku što dalje
- Enkapsulira kompleksnost algoritma pa se korisnik može posvetiti samo implementaciji odgovarajuće funkcije (tj. implementaciji sučelja *BiFunction*)

```
ava   Map.class    
  
default V compute(K key,  
    BiFunction<? super K, ? super V, ? extends V> remappingFunction) {  
    Objects.requireNonNull(remappingFunction);  
    V oldValue = get(key);  
  
    V newValue = remappingFunction.apply(key, oldValue);  
    if (newValue == null) {  
        // delete mapping  
        if (oldValue != null || containsKey(key)) {  
            // something to remove  
            remove(key);  
            return null;  
        } else {  
            // nothing to do. Leave things as they were.  
            return null;  
        }  
    } else {  
        // add or replace old mapping  
        put(key, newValue);  
        return newValue;  
    }  
}
```

# *compute* u sučelju *Map* – primjer korištenja (1)

11\_InnerNestedLambda/hr/fer/oop/defmethods/ExampleMapCompute.java

- Brojanje pojavljivanja pojedinog tipa vozila
  - Implementacija anonimnom klasom

```
Map<CarType, Integer> carTypesCount = new HashMap<>();
List<Car> cars = CarCatalog.loadCars();
cars.forEach(new Consumer<Car>() {
    @Override
    public void accept(Car t) {
        Integer newVal = carTypesCount.compute(t.getType(),
            new BiFunction<CarType, Integer, Integer>() {
                @Override
                public Integer apply(CarType key, Integer value){
                    return value == null ? 1 : value + 1;
                }
            });
        System.out.printf("%s raises number of %s cars to %d %n",
            t.getName(), t.getType(), newVal);
    }
});
```

## *compute i super i extends (1)*

- U primjerima su korišteni točno traženi tipovi

```
Integer newVal = carTypesCount.compute(t.getType(),
    new BiFunction<CarType, Integer, Integer>() {
        @Override
        public Integer apply(CarType key, Integer value) {
            return value == null ? 1 : value + 1;
        }
    });
```

- Kad bi se *Integer* mogao naslijediti mogla bi postojati hijerarhija  
*Object – Number – Integer – MyInt*
  - Tada bi sljedeći kod također bio ispravan

```
Integer newVal = carTypesCount.compute(t.getType(),
    new BiFunction<CarType, Number, MyInt>() {
        @Override
        public MyInt apply(CarType key, Number value) {
            ...
        }
    });
```

# compute i super i extends (2)

- PECS

“Producer extends  
Consumer super”


- compute šalje ključ i vrijednost tipa K i V nekoj metodi koja to mora biti u stanju primiti

- Vraćena (proizvedena) vrijednosti se mora moći pretvoriti u V

```
ava Map.class
default V compute(K key,
    BiFunction<? super K, ? super V, ? extends V> remappingFunction) {
    Objects.requireNonNull(remappingFunction);
    V oldValue = get(key);

    V newValue = remappingFunction.apply(key, oldValue);
    if (newValue == null) {
        // delete mapping
        if (oldValue != null || containsKey(key)) {
            // something to remove
            remove(key);
            return null;
        } else {
            // nothing to do. Leave things as they were.
            return null;
        }
    } else {
        // add or replace old mapping
        put(key, newValue);
        return newValue;
    }
}
```

```
Integer newVal = carTypesCount.compute(t.getType(),
    new BiFunction<CarType, Number, MyInt>() {
        @Override
        public MyInt apply(CarType key, Number value) {
```



# *compute* u sučelju *Map* – primjer korištenja (1)

- Brojanja pojavljivanja pojedinog tipa vozila
  - Implementacija lambda klasom

11\_InnerNestedLambda hr/fer/oop/defmethods/ExampleMapComputeLambda.java

```
Map<CarType, Integer> carTypesCount = new HashMap<>();
List<Car> cars = CarCatalog.loadCars();
cars.forEach(car -> {
    Integer newVal = carTypesCount.compute(car.getType(),
        (key, value) -> value == null ? 1 : value + 1);
    System.out.printf("%s raises number of %s cars to %d %n",
        car.getName(), car.getType(), newVal);
});
```

# Metoda *merge* u sučelju *Map*

- Metoda *merge* je slična metodi *compute*
  - Ako ključ (prvi argument) nije u mapi ili mu je pridružen *null*, pridružuje mu se vrijednost zadana kao drugi argument
  - Inače, pridružuje mu se vrijednost izračunata temeljem funkcije (*BiFunction*) zadane kao treći argument. Ako je rezultat poziva funkcije *null*, ključ se uklanja iz mape

11\_InnerNestedLambda/hr/fer/oop/defmethods/ExampleMapMergeLambda.java

```
Map<CarType, Integer> carTypesCount = new HashMap<>();  
List<Car> cars = CarCatalog.loadCars();  
cars.forEach(car -> carTypesCount.merge(car.getType(), 1,  
                                         (oldValue, value) -> oldValue + value));
```