

# Lambda izrazi

---

# Sadržaj

---

Povijest razvoja višenitnosti u Javi

Zašto koristiti Lambda izraze?

Sintaksa lambda izraza

Funkcionalna sučelja

Vrijednosti u lambda izrazima

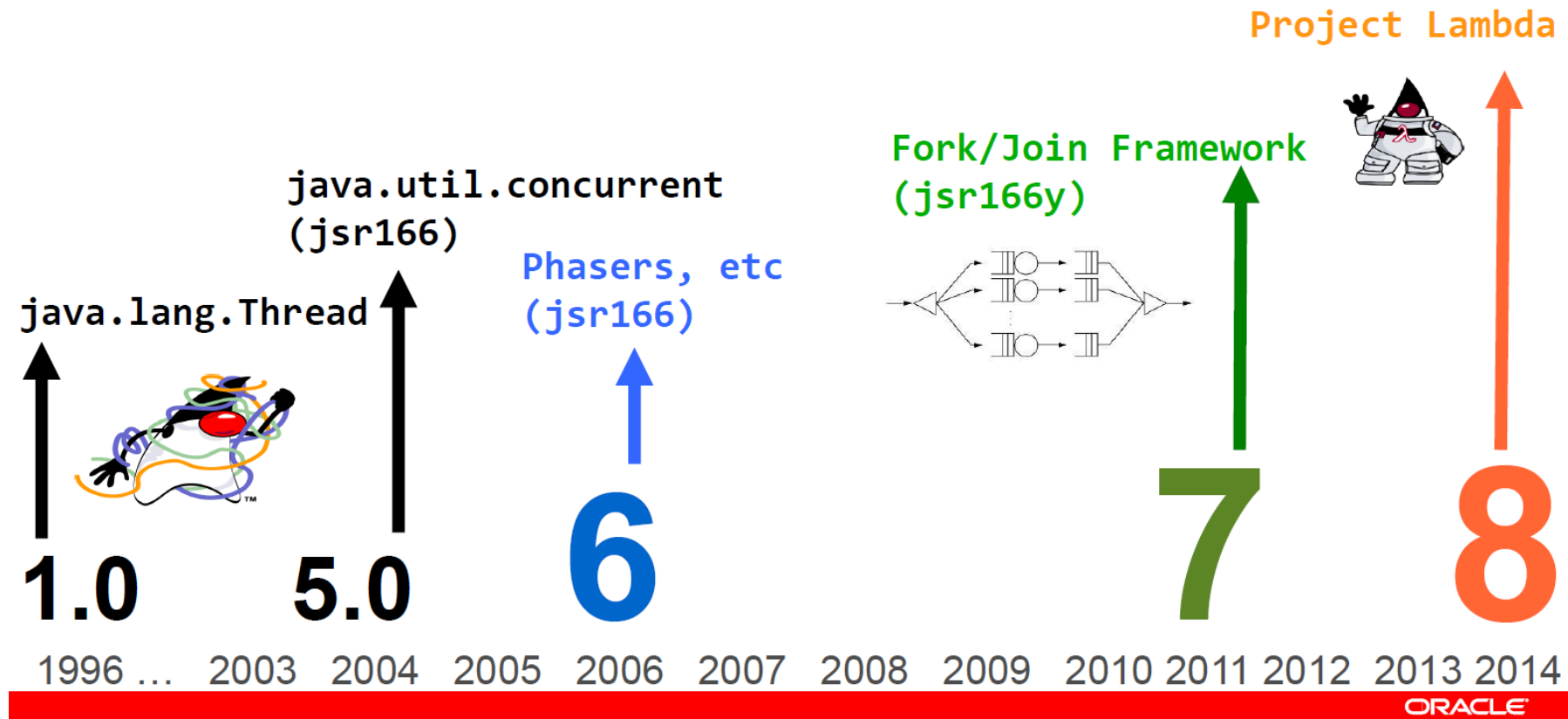
Korištenje tipova parametara

Tokovi

Optional

Reference za metode

# Povijest razvoja višenitnosti u Javi



# Zašto koristiti lambda izraze?

---

- Bez lambda izraza u radu sa zbirkama potrebno je koristiti vanjsku iteraciju (engl. *external iteration*):

```
List<Student> students = ...  
double highestScore = 0.0;  
  
for (Student s : students) {  
    if (s.getGradYear() == 2011) {  
        if (s.getScore() > highestScore)  
            highestScore = s.getScore();  
    }  
}
```

- Iteracija se kontrolira kroz programski kod i prolazi od početka do kraja zbirke
- Nije *thread-safe*

# Zašto koristiti lambda izraze?

---

- Korištenjem lambda izraza navedeni programski kod bi izgledao ovako:

```
double highestScore = students.stream()
    .filter(new Predicate<Student>() {
        public boolean op(Student s) {
            return s.getGradYear() == 2011;
        }
    })
    .map(new Mapper<Student, Double>() {
        public Double extract(Student s) {
            return s.getScore();
        }
    })
    .max();
```

- Iteracija se izvodi od strane *librarya*
- Prolazak po elementima se može paralelizirati i u jednom prolazu
- *Thread-safe*
- Teže za razumijevanje i korištenje

# Zašto koristiti lambda izraze?

---

- Navedeni programski kod može se još skratiti na sljedeći način:

```
List<Student> students = ...
```


```
double highestScore = students.stream()  
    .filter(Student s -> s.getGradYear() == 2011)  
    .map(Student s -> s.getScore())  
    .max();
```

- Navedeni izraz je čitljiviji, apstraktniji i manje podložen pogreškama

# Sintaksa lambda izraza

---

- Lambda izrazi imaju sljedeću sintaksu:

 Lambda operator

`(parameters) -> { lambda-body }`

- Tijelo lambde može baciti iznimku
- Ako se tijelo sastoji samo od jedne linije, nije potrebno koristiti zagrade
- Nije nužno korištenje eksplicitne „return” naredbe
- Ako se koristi samo jedan parametar, nisu potrebne zagrade
- Ako nema parametara, koriste se samo zagrade

# Funkcionalna sučelja

---

- Sadrže samo jednu apstraktnu metodu, npr.:

```
interface FileFilter      { boolean accept(File x); }  
interface ActionListener { void actionPerformed(...); }  
interface Callable<T>    { T call(); }
```

- Lambda izrazi predstavljaju implementaciju te apstraktne metode



# Funkcionalna sučelja

---

- Još neka važna sučelja su navedena u sljedećoj tablici:

Interface name	Arguments	Returns	Example
<code>Predicate&lt;T&gt;</code>	<code>T</code>	<code>boolean</code>	Has this album been released yet?
<code>Consumer&lt;T&gt;</code>	<code>T</code>	<code>void</code>	Printing out a value
<code>Function&lt;T,R&gt;</code>	<code>T</code>	<code>R</code>	Get the name from an <code>Artist</code> object
<code>Supplier&lt;T&gt;</code>	<code>None</code>	<code>T</code>	A factory method
<code>UnaryOperator&lt;T&gt;</code>	<code>T</code>	<code>T</code>	Logical not (!)
<code>BinaryOperator&lt;T&gt;</code>	<code>(T, T)</code>	<code>T</code>	Multiplying two numbers (*)

# Funkcionalna sučelja - Predicate

---

- „Predicate” predstavlja „Boolean” vrijednost funkcije s jednim argumentom
- Sadrži korisne statičke metode poput „and()”, „or()”, „negate()” i „isEqual()”
- Primjeri:

```
Student s -> s.graduationYear() == 2011
```

```
Files.find(start, maxDepth,  
    (path, attr) -> String.valueOf(path).endsWith(".js") &&  
        attr.size() > 1024,  
    FileVisitOption.FOLLOW_LINKS);
```

# Funkcionalna sučelja – Consumer<T>

---

- Operacija koja prima jednu vrijednost i ne vraća rezultat
- Omogućava ulančavanje funkcija korištenjem metode „andThen(Consumer after)”
- Primjeri:

```
String s -> System.out.println(s)
```

```
(k, v) -> System.out.println("key:" + k + ", value:" + v)
```

# Funkcionalna sučelja – Function<T>

---

- Prima jedan argument i vraća rezultat
- Može primiti i vraćati različite tipove parametara
- Omogućava korištenje statičkih metoda za ulančavanje: „compose” i „andThen”
- Primjer:

```
Student s -> s.getName()
```

```
(String name, Student s) -> new Teacher(name, student)
```

# Funkcionalna sučelja – UnaryOperator<T>

---

- Specijalizirani oblik „Function” sučelja
- Prima jedan parametar i vraća isti tip parametra po principu:  
    `T apply(T a)`
- Primjer:

```
String s -> s.toLowerCase()
```

# Funkcionalna sučelja – BinaryOperator<T>

---

- Specijalizirani oblik „Function” sučelja
- Prima dva parametra i vraća isti tip parametra po principu:

`T apply(T a, T b)`

- Primjer:

```
(String x, String y) -> {  
    if (x.length() > y.length())  
        return x;  
    return y;  
}
```

# Vrijednosti u lambda izrazima

---

- Slično kao i kod ugniježđenih anonimnih funkcija, parametri lambda funkcije se ponašaju kao da imaju „final” modifikator
- Modifikator „final” se može izostaviti, ali se podrazumijeva
- U sljedećem primjeru varijabli „name” je implicitno dodijeljen modifikator „final”:

```
String name = getUsername();  
button.addActionListener(event -> System.out.println("hi " + name));
```

- Često se govori i o izrazu „effectively final” koji podrazumijeva da se varijabli samo jednom može dodijeliti vrijednost ako se koristi u lambda izrazu, kao u prijašnjem primjeru

# Vrijednosti u lambda izrazima

---

- Ako se npr. pokuša prevesti sljedeći programski isječak:

```
String name = getUsername();  
name = formatUserName(name);  
button.addActionListener(event -> System.out.println("hi " + name));
```

- Kompajler u tom slučaju dojavljuje pogrešku:

local variables referenced from a lambda expression  
must be **final** or effectively **final**.



# Vrijednosti u lambda izrazima

---

- U slučaju korištenja lokalnih varijabli, nužno je da budu „*effective final*”:

```
void expire(File root, long before) {  
    root.listFiles(File p -> p.lastModified() <= before);  
}
```

# Korištenje tipova parametara

---

- Tipovi parametara se često mogu izostavljati radi čitljivosti, ali ponekad ih je bolje pisati iako nije nužno, kako bi programski kod bio razumljiviji
- Mogućnost izostavljanja tipova parametara je već uvedena od Jave 7 korištenjem „diamond operatora”, npr.:

```
Map<String, Integer> oldWordCounts = new HashMap<String, Integer>();  
Map<String, Integer> diamondWordCounts = new HashMap<>();  
useHashmap(new HashMap<>());  
...  
private void useHashmap(Map<String, String> values);
```

# Korištenje tipova parametara

---

- Na primjer, funkcijsko sučelje „Predicate” izgleda ovako:

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

- Može se koristiti na sljedeći način:

```
Predicate<Integer> atLeast5 = x -> x > 5;
```

- Slično je moguće koristiti i sljedeće funkcijsko sučelje:

```
BinaryOperator<Long> addLongs = (x, y) -> x + y;
```

- Ako se ne navede parametar (npr. „Long”) kompajler neće moći zaključiti s kakvim tipovima podataka se obavlja binarna operacija i dojaviti će pogrešku

# Tokovi

---

- Tokovi (engl. *Streams*) omogućavaju pisanje programskog koda za obradu podataka u zbirkama (engl. *collections*) na višoj razini apstrakcije
- Sučelje „Stream” sadrži niz metoda koje je moguće izvršavati nad zbirkama podataka korištenjem višejezgrenih procesora
- Na primjer, ako je bez korištenja tokova potrebno izračunati koliko objekata iz zbirke ispunjava određeni uvjet, to se može izvršiti ovako:

```
int count = 0;
for (Artist artist : allArtists) {
    if (artist.isFrom("London")) {
        count++;
    }
}
```

# Tokovi

---

- *Stream pipeline* se sastoji od tri elementa:
  - Izvora
  - Nula ili više „intermediate” operacija
  - „Terminal” operacije



# Tokovi

---

- Primjer:

Source

Intermediate operations

```
int total = transactions.stream()  
    .filter(t -> t.getBuyer().getCity().equals("London"))  
    .mapToInt(Transaction::getPrice)  
    .sum();
```

Terminal operation

The diagram illustrates the classification of Java Stream operations. It shows a code snippet for calculating the total price of transactions in London. Red arrows indicate the following classifications:

- Source:** Points to the `stream()` method, which creates a stream from the `transactions` collection.
- Intermediate operations:** Points to the `filter()` and `mapToInt()` methods, which transform the stream into another stream.
- Terminal operation:** Points to the `sum()` method, which triggers the execution of the stream and returns the final result.

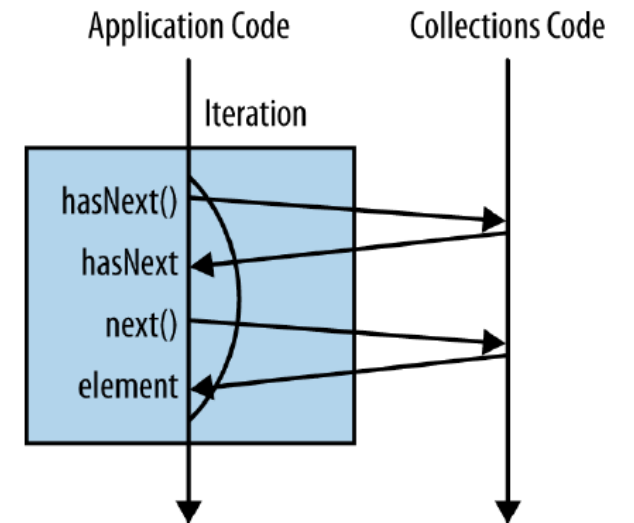
# Tokovi

---

- U slučaju korištenja iteratora, to izgleda ovako:

```
int count = 0;
Iterator<Artist> iterator = allArtists.iterator();
while(iterator.hasNext()) {
    Artist artist = iterator.next();
    if (artist.isFrom("London")) {
        count++;
    }
}
```

- Takav način iteriranja po elementima zbirke naziva se „vanjska iteracija”:



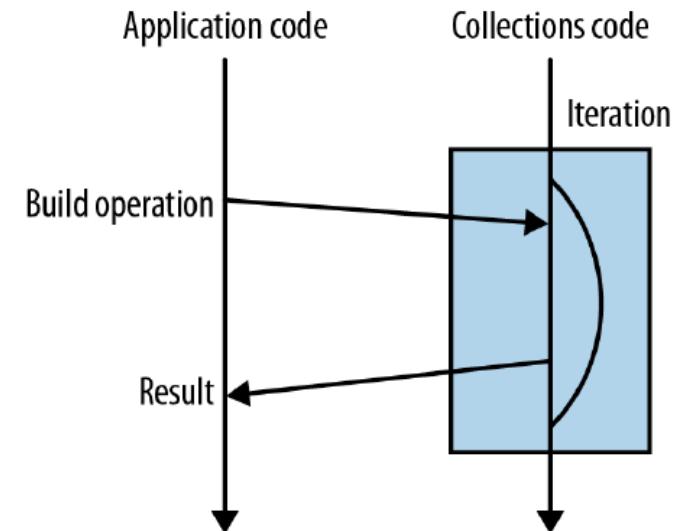
# Tokovi

---

- Korištenjem tokova programski kod svodi se na ovo:

```
long count = allArtists.stream()  
    .filter(artist -> artist.isFrom("London"))  
    .count();
```

- Metoda „stream” vraća sučelje tipa „Stream” koje sadrži metode za obavljanje kompleksnih operacija nad zbirkama korištenjem interne iteracije:





# Tokovi

---

- Na početku se pozivanjem metode „filter” postavlja „kriterij” po kojem će se dohvaćati samo određeni elementi iz zbirke (ne kreira se nova „filtrirana” zbirka)
- Takve operacije se često kategoriziraju kao „lazy” (ne izvršavaju se odmah)
- Metoda „count” na kraju pobrojava sve objekte iz zbirke koji su „prošli filtriranje”
- Takve operacije se kategoriziraju kao „eager” (odmah se izvršavaju) i „okidaju” izvođenje svih ostalih „lazy” operacija koje su prethodno izvršene

# Tokovi

---

- Na primjer, sljedeći programski kod ne ispisuje ništa:

```
allArtists.stream()
    .filter(artist -> {
        System.out.println(artist.getName());
        return artist.isFrom("London");
    });
```

- Tek dodavanjem završne (engl. *terminal*) operacije „count” pokreću se sve međuoperacije (engl. *intermediate*) kao što je filter:

```
long count = allArtists.stream()
    .filter(artist -> {
        System.out.println(artist.getName());
        return artist.isFrom("London");
    })
    .count();
```

# Tokovi

---

- Osnovna razlika između „lazy” i „eager” operacija je u tome što one vraćaju: ako metoda vraća „Stream”, onda je „lazy”, a ako ne vraća ništa, onda je „eager”
- Korištenjem tokova je moguće „ulančavanje” različitih poziva operacija, pri čemu se na kraju samo jednom prolazi kroz cijelu zbirku

# Optional

---

- Predstavlja novi tip podatka koji predstavlja bolju alternativu od „null” vrijednosti – vrijednost koja može postojati ili ne
- Često je korisna u slučaju kad „filter” ne vrati nikakvu vrijednost (kad je „Stream” prazan)
- Primjeri korištenja:

```
Optional<String> a = Optional.of("a");  
assertEquals("a", a.get());  
Optional emptyOptional = Optional.empty();  
Optional alsoEmpty = Optional.ofNullable(null);  
assertFalse(emptyOptional.isPresent());  
assertTrue(a.isPresent());
```

# Optional

---

- Omogućava izbjegavanje problema s iznimkom „NullPointerException” pa se umjesto ovoga:

```
String direction = gpsData.getPosition().getLatitude().getDirection();
```

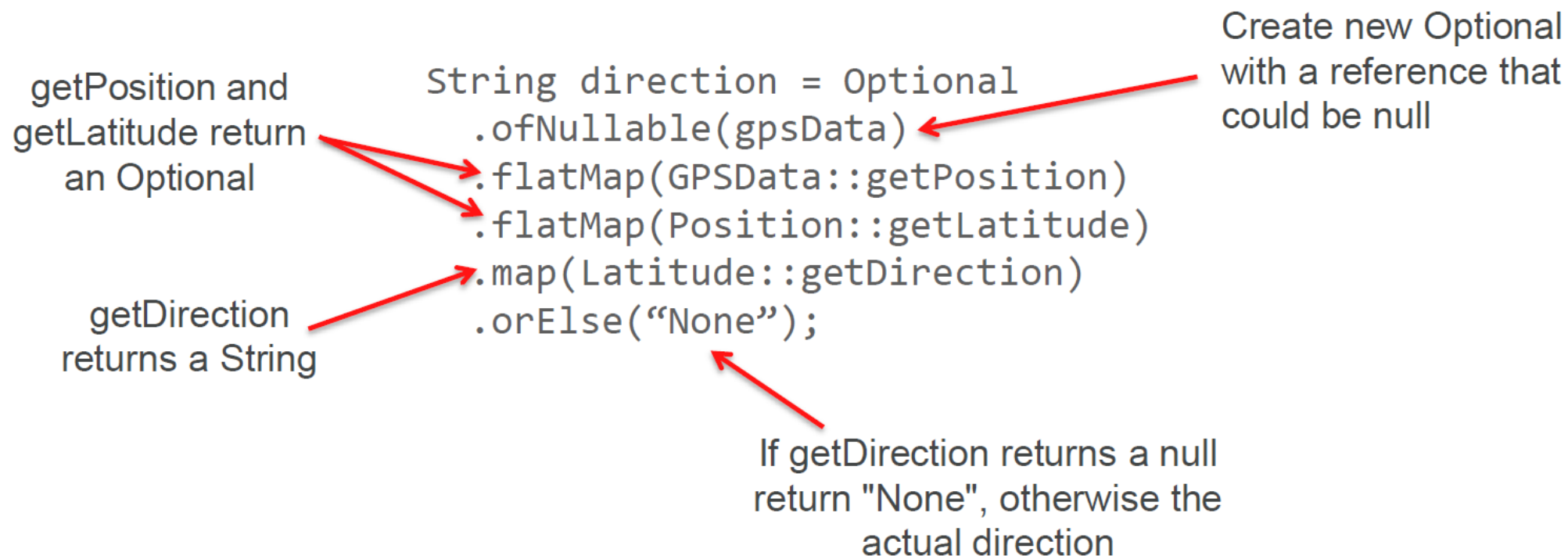
```
String direction = "UNKNOWN";
```

```
if (gpsData != null) {  
    Position p = gpsData.getPosition();  
  
    if (p != null) {  
        Latitude latitude = p.getLatitude();  
  
        if (latitude != null)  
            direction = latitude.getDirection();  
    }  
}
```

# Optional

---

- Može se koristiti ovo:



# Optional

---

- U jednostavnijim oblicima se može koristiti i na ovaj način:

```
if (x != null) {  
    print(x);  
}
```

```
opt.ifPresent(x -> print(x));  
opt.ifPresent(this::print);
```

# Java 9 proširenja Optional tipa

---

- U Javi 9 dodane su nove mogućnosti korištenja Optional tipa podataka, na primjer, moguće je koristiti referencu na metodu „Optional::stream” ili još kraću alternativu:

```
public Stream<Customer> findCustomers(Collection<String> customerIds) {  
    return customerIds.stream()  
        .map(this::findCustomer)  
        .flatMap(Optional::stream)  
}
```

```
public Stream<Customer> findCustomers(Collection<String> customerIds) {  
    return customerIds.stream()  
        .flatMap(id -> findCustomer(id).stream());  
}
```



# Java 9 proširenja Optional tipa

---

- Također je moguće vraćati listu vrijednosti prema zadanom kriteriju:

```
public List<Order> findOrdersForCustomer(String customerId) {  
    return findCustomer(customerId)  
        .map(this::getOrders)  
        .orElse(new ArrayList<>());  
}
```

```
public Stream<Order> findOrdersForCustomer(String customerId) {  
    return findCustomer(customerId)  
        .stream()  
        .map(this::getOrders)  
        .flatMap(List::stream);  
}
```

# Java 9 proširenja Optional tipa

---

- Metodom „Optional::or” može se definirati nekoliko kriterija pretrage pri čemu svaki od njih može vratiti praznu listu radi čega je potrebno koristiti povratni tip „Optional”:

```
public Optional<Customer> findCustomer(String customerId) {  
    return customers.findInMemory(customerId)  
        .or(() -> customers.findOnDisk(customerId))  
        .or(() -> customers.findRemotely(customerId));  
}
```

# Java 9 proširenja Optional tipa

---

- Dodane su metode „ifPresentOrElse”, „ifPresent” i „ifEmpty” koje se mogu koristiti na sljedeći način:

```
public void logLogin(String customerId) {  
    findCustomer(customerId)  
        .ifPresentOrElse(  
            this::logLogin,  
            () -> logUnknownLogin(customerId)  
        );  
}
```

```
public void logLogin(String customerId) {  
    findCustomer(customerId)  
        .ifPresent(this::logLogin)  
        .ifEmpty(() -> logUnknownLogin(customerId));  
}
```

# Reference za metode

---

- Omogućavaju ponovno iskorištavanje metoda kao lambda izraza:

```
FileFilter x = File f -> f.canRead();
```



```
FileFilter x = File::canRead;
```

- Pišu se po principu „referenca objekta :: naziv metode”
- Mogu biti statičke metode, metode klase ili metode objekta

# Reference za metode

---

- Pravila za korištenje:

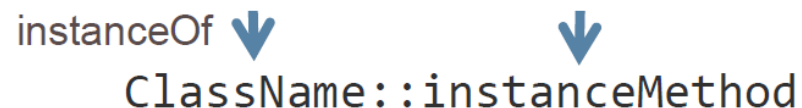
Lambda  
Method Ref

`(args) -> ClassName.staticMethod(args)`

  
`ClassName::staticMethod`

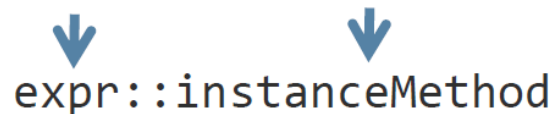
Lambda  
Method Ref

`(arg0, rest) -> arg0.instanceMethod(rest)`

  
`arg0::instanceMethod`

Lambda  
Method Ref

`(args) -> expr.instanceMethod(args)`

  
`expr::instanceMethod`


# Reference za metode

---

- Primjeri:


Lambda  
Method Ref

```
(String s) -> System.out.println(s)
```

  
System.out::println


Lambda  
Method Ref

```
(String s, int i) -> s.substring(i)
```

  
String::substring

Lambda  
Method Ref

```
Axis a -> getLength(a)
```

  
this::getLength

# Pitanja?

---