

The background of the slide is a deep blue gradient. On the left side, there is a stylized, semi-transparent globe showing latitude and longitude lines. Several white, wavy, line-like patterns, resembling stylized waves or data paths, are overlaid on the globe and extend towards the right. The overall aesthetic is modern and technological.

Lambda izrazi

Programski jezici II

Prosljeđivanje funkcija

- Mnogi programski jezici dozvoljavaju prosljeđivanje funkcija kao parametara metoda
 - Dinamički i obično slabo tipizirani:
 - JavaScript, Lisp, Scheme,...
 - Snažno tipizirani
 - Ruby, Scala, Clojure, ML,...

Lambda izrazi

- Koncizna sintaksa
 - Jezgrovitiji i čistiji pristup u poređenju sa anonimnim unutrašnjim klasama
- Nedostaci anonimnih unutrašnjih klasa
 - Glomazne, konfuzija pri korišćenju ključne riječi „this“, nema pristupa lokalnim promjenjivim koje nisu final, teško za optimizovati, ...
- Pogodne za korišćenje sa Stream API-jem
- Programerima je koncept poznat
 - Callback, map/reduce idiom, ...

Lambda izrazi

- „Klasični“ pristup

```
button1.addActionListener(new ActionListener() {  
    @Override  
    public void actionPerformed(ActionEvent event) {  
        setBg(Color.BLUE);  
    }  
});
```

- „Novi“ pristup

```
button1.addActionListener(event -> setBg(Color.BLUE));
```

Lambda izrazi

- Ohrabruje se upotrebu funkcionalnog programiranja
 - Mnoge klase problema se lakše rješavaju upotrebom funkcionalnog programiranja i rezultiraju u programskom kodu koji je jasniji i jednostavniji za održavanje
- Funkcionalno programiranje ne zamjenjuje OOP – OOP je i dalje osnovni pristup za predstavljanje tipova, dok funkcionalno programiranje može poboljšati mnoge metode i algoritme
- Podržava se upotreba stream-ove
 - Stream-ovi su wrapper-i oko data source-va (nizovi, kolekcije, itd.) koji koriste lambde, podržavaju map/filter/reduce, koriste lazy evaluaciji i mogu se učiti paralelnim (automatski)
 - „Klasični“ pristup

```
for (Employee e: employees) {  
    e.doSomething();  
}
```
 - „Novi“ pristup

```
employees.stream().parallel().forEach(e -> e.doSomething());
```

Lambda izrazi

- Java 7 primjer

```
Arrays.sort(testStrings, new Comparator<String>() {  
    @Override  
    public int compare(String s1, String s2) {  
        return(s1.length() - s2.length());  
    }  
});
```

- Java 8 primjer

```
Arrays.sort(testStrings,  
    (s1, s2) -> s1.length() - s2.length());
```

Lambda izrazi

- Java 7 primjer

```
Arrays.sort(testStrings, new Comparator<String>() {  
    @Override  
    public int compare(String s1, String s2) {  
        return(s1.length() - s2.length());  
    }  
});
```

- Java 8 primjer

```
Arrays.sort(testStrings,  
    (String s1, String s2) -> {return (s1.length() - s2.length()); });
```

- Ideja

- Iz API-ja, Java „zna“ da je Comparator drugi argument metode Arrays.sort, tako da to nije potrebno naglašavati
- Comparator ima samo jednu metodu, tako da nije potrebno naglašavati naziv metode – compare
- Dodaje se “->” između parametara metode i tijela metode

Lambda izrazi

- Java 7 primjer

```
Arrays.sort(testStrings, new Comparator<String>() {  
    @Override  
    public int compare(String s1, String s2) {  
        return(s1.length() - s2.length());  
    }  
});
```

- Java 8 primjer

```
Arrays.sort(testStrings,  
(s1, s2) -> {return (s1.length() - s2.length()); });
```

- Ideja

- Gledajući pri argument metode sort (testStrings), Java može zaključiti da je tip drugog argumenta Comparator<String>, te da su, prema tome, parametri compare metode String-ovi – iz tog razloga nije potrebno naglašavati tipove parametara metode compare
- Java i dalje radi snažnu provjeru tipova u vrijeme kompajliranja – razlika je u tome što kompajler zaključuje o kojim tipovima se radi – slično kao sa diamond operatorom

```
List<String> abc = new ArrayList<>();
```
- U slučaju da su tipovi dvosmisleni, kompajler će prijaviti da ne može zaključiti o kojim tipovima se radi – u ovom slučaju tipove je potrebno navoditi

Lambda izrazi

- Java 7 primjer

```
Arrays.sort(testStrings, new Comparator<String>() {  
    @Override  
    public int compare(String s1, String s2) {  
        return s1.length() - s2.length();  
    }  
});
```

- Java 8 primjer

```
Arrays.sort(testStrings, (s1, s2) -> s1.length() - s2.length());
```

- Ideja

- Ako tijelo metode čini jedan return izraz, vitičaste zagrade i „return“ mogu da se odbace
- Ove se ne može uvijek uraditi, pogotovo ako se koriste petlje ili if naredbe
- Lambde se obično koriste kada je tijelo metode kratko, tako da je ovo obično moguće uraditi

Lambda izrazi

- Ako metoda interfejsa ima tačno jedan parametar i male zagrade je moguće odbaciti
- Java 7 primjer

```
button.addActionListener(new ActionListener() {  
    @Override  
    public void actionPerformed(ActionEvent e) {  
        doSomethingWith(e);  
    }  
});
```

- Java 8 primjer – sa malim zagradama

```
button.addActionListener((e) -> doSomethingWith(e));
```

- Java 8 primjer – bez malih zagrada

```
button.addActionListener(e -> doSomethingWith(e));
```

- U praksi – stvara se objekat anonimne klase koja implementira interfejs sa jednom metodom

Lambda izrazi

- Funkcionalni interfejsi – interfejsi koji imaju jednu metodu
- Drugi naziv – SAM (Single Abstract Method) interfejsi
- `@FunctionalInterface`
 - Detektuje greške za vrijeme kompajliranja
 - Ako se naknadno doda druga apstraktna metoda u interfejs, interfejs se neće moći kompajlirati
 - Izražava namjeru predviđenu dizajnom
 - „govori“ programerima da će se za ovaj interfejs koristiti lambde
 - Nije obavezan
 - Isto kao i `@Override`
 - Lambde se mogu koristiti svuda gdje se očekuju interfejsi sa jednom apstraktnom metodom (funkcionalni interfejsi, SAM interfejsi), bez obzira da li taj interfejs koristi `@FunctionalInterface`

Lambda izrazi

- Opšti oblik definicije lambda izraza: (parametri) -> {tijelo}
 - parametri – uobičajena lista tipova i imena
 - tipovi mogu da se izostave, ako se o njima može zaključiti iz konteksta
 - moraju se ili navesti svi tipovi ili izostaviti svi tipovi
 - ako u listi postoji samo 1 parametar bez tipa – zagrade () mogu da se izostave
 - zagrade () su obavezne ako nema parametara
 - tijelo – uobičajeno tijelo funkcije (blok koji koristi parametre)
 - ako je samo jedna naredba (return), mogu da se izostave zagrade {} i reč return
 - tada se tijelo svodi na izraz koji izračunava vrijednost lambda izraza
 - tip lambda izraza – ne navodi se
 - o njemu se zaključuje na osnovu tipa izraza u return naredbi
 - tip može biti i void

Reference metoda

- Statičke metode
 - Umjesto (args) -> `ClassName.staticMethodName(args)`
 - Koristi se `ClassName::staticMethodName`
 - `Math::cos`, `Arrays::sort`, `String::valueOf`
 - Za funkcije koje imaju ime, nije potrebno pisati lambda izraz – moguće je koristiti ime metode
 - Potpis metode mora odgovarati potpisu metode funkcionalnog interfejsa
 - `Math::cos`
 - `x -> Math.cos(x)`
- `variable::instanceMethod`
 - `someString::toUpperCase`
 - `() -> someString.toUpperCase()`
- `Class::instanceMethod`
 - `String::toUpperCase`
 - `s -> s.toUpperCase()`
- `ClassOrType::new`
 - `Employee::new`
 - `() -> new Employee()`

Reference metoda

- `m1(Math::cos)`
- `m1(d -> Math.cos(d))`
- `m2(System.out::println)`
- `m2(s -> System.out.println(s))`
- `m3(Class::twoArgMethod)`
- `m3((a, b) -> Class.twoArgMethod(a, b))`

Vidljivost varijabli u Lambda izrazima

- Lambde ne uključuju novi nivo vidljivosti
 - This promjenljiva referencira vanjsku klasu, ne anonimnu unutrašnju klasu u koju je Lambda uključena
 - Ne postoji “OuterClass.this” promjenjiva, osim ako se Lambda ne nalazi u običnoj unutrašnjoj klasi
 - Ne mogu „uvesti“ ime nove promjenjive koje je identično imenu promjenjive u metodi koja kreira lambdu
 - Lambde mogu referencirati (ne i modifikovati) lokalne varijable iz okružujuće metode
 - I dalje mogu referencirati i modifikovati promjenjive instance iz okružujuće klase