Funkcionální prvky, lambda výrazy

O co jde?

- Java byla navržena jako čistě objektový jazyk.
- Lambda výrazy do tohoto jazyka přináší prvky funkcionálního programování.
- Jak mohou tyto dva různé světy koexistovat?
- Proč a jak je funkcionální syntaxe integrována do objektového API Javy?

Motivační příklad

Založeno na Java Tutorial

- Předpokládejme, že chceme vytvořit sociální síť. Administrátorům chceme umožnit provádění různých akcí jako například zasílání zpráv těm uživatelům sociální sítě, kteří splňují nějaká kritéria.
- Předpokládejme, že uživatelé jsou reprezentování následující třídou:

```
public class Person {
    public enum Sex {
        MALE, FEMALE
    }
    private String name;
    private LocalDate birthday;
    private Sex gender;
    private String emailAddress;

    public int getAge() {
        // ...
    }
    public void printPerson() {
        // ...
    }
}
```

Krok 1: Primitivní řešení

- Dále předpokládejme, že jsou uživatelé sociální sítě uložení v seznamu `List<Person>`.
- Základní implementace vyhledávání lidí podle kritéria může vypadat následovně:

```
public static void printPersonsOlderThan(List<Person> roster, int age) {
    for (Person p : roster) {
```

```
        if (p.getAge() >= age) {
            p.printPerson();
        }
    }
}
```

Otázky

Q

Co když chceme přidat další operaci, např. výpis lidí mladších než nějaký věk?

A

Musí se přidat nová metoda, nebo se existující metoda musí pojmout víc obecněji.

Krok 2: Zobecněné vyhledávání

```
public static void printPersonsWithinAgeRange(List<Person> roster, int low, int high)
{
    for (Person p : roster) {
        if (low <= p.getAge() && p.getAge() < high) {
            p.printPerson();
        }
    }
}
```

Otázky

Q

Co když chceme vypisovat uživatele specifického pohlaví, nebo dokonce kombinace věk a pohlaví?

Q

Co když se rozhodneme změnit třídu `Person` a přidat do ní atributy, např. vzájemné vztahy nebo geografická lokace?

A

Přestože je tato metoda obecnější než předchozí `printPersonsOlderThan`, snaha vytvořit specifickou metodu pro každý možný vyhledávací dotaz je neudržitelná.

Řešení

Oddělit kód, který specifikuje vyhledávací kritéria, od samotného vyhledávání.

Jména metod musí být výstižná a popisná. Proto byla metoda přejmenována.

Krok 3: Použití vlastního rozhraní [1/2]

- Definujeme rozhraní pro vyhledávací a vytvoříme implementaci:

```
interface CheckPerson {
    boolean test(Person p);
}

class CheckPersonEligibleForSelectiveService implements CheckPerson {
    public boolean test(Person p) {
        return p.gender == Person.Sex.MALE &&
            p.getAge() >= 18 &&
            p.getAge() <= 25;
    }
}
```

Ve stejnou chvíli můžeme mít definováno několik vyhledávacích kritérií (tříd implementujících rozhraní).

Krok 3: Použití vlastního rozhraní [2/2]

- Vyhledávací metoda se změní následovně:

```
public static void printPersons(List<Person> roster, CheckPerson tester) {
    for (Person p : roster) {
        if (tester.test(p)) {
            p.printPerson();
        }
    }
}
```

A volá se takto:

```
List<Person> roster = ...
printPersons(roster, new CheckPersonEligibleForSelectiveService());
```

Q

Je nutné definovat `CheckPersonEligibleForSelectiveService` ve speciální třídě?

A

Není. Můžeme použít anonymní třídy a redukovat tak kód.

Krok 4: Použití anonymní třídy

```
interface CheckPerson {
    boolean test(Person p);
}
// separate iteration and tester
public static void printPersons(List<Person> roster, CheckPerson tester) {
    for (Person p : roster) {
        if (tester.test(p)) {
            p.printPerson();
        }
    }
}
```

Provedení filtrace

```
// do the filtering
List<Person> roster = ...
printPersons(
    roster,
    new CheckPerson() {
        public boolean test(Person p) {
            return p.getGender() == Person.Sex.MALE
                && p.getAge() >= 18
                && p.getAge() <= 25;
        }
    }
);
```

- Všimněte si, že `CheckPerson` je funkcionální - obsahuje jedinou metodu. Proto je ale název metody nepodstatný.

  Q

  Mohli bychom název metody nějak vynechat?

  A

  Ano, pokud použijeme **lambda výraz**, který se dá chápat jako definice **anonymní metody** implementující nějaké **funkcionální rozhraní**.

Krok 5: Použití lambda výrazu

```
List<Person> roster = ...
printPersons(
    roster,
```

```
    (Person p) -> p.getGender() == Person.Sex.MALE
        && p.getAge() >= 18
        && p.getAge() <= 25
);
```

Vysvětlení lambda syntaxe →

- Levá část (před šipkou) obsahuje vstupní parametry, pravá strana pak kód případně výstupní hodnotu.
- Všimněte si, že rozhraní `CheckPerson` se teď v kódu objevuje pouze typ argumentu v metodě `printPerson()`.
- Ale název typu (ani metody, jak už víme) není důležitý.

Q

Mohli bychom vynechat z kódu i typ `CheckPerson`?

A

Ano. Java nabízí pro takové případy vlastní generická předdefinovaná rozhraní.

Krok 6: Použití existujících funkcionálních rozhraní [1/3]

- Zamysleme se nad původní definicí našeho rozhraní:

```
interface CheckPerson {
    boolean test(Person p);
}
```

Zobecnění pomocí generických typů

- Jeho význam můžeme zobecnit pomocí generických typů podobně, jako je tomu u rozhraní `java.util.functions.Predicate`:

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

Krok 6: Použití existujících funkcionálních rozhraní [2/3]

- `CheckPerson` tedy již nadále nepotřebujeme. Můžeme místo něj použít `Predicate<T>`:

```
public static void printPersons(List<Person> roster, Predicate<Person> tester) {
    for (Person p : roster) {
        if (tester.test(p)) {
            p.printPerson();
        }
    }
}
```

Vlastní filtrace zůstává

```
List<Person> roster = ...
printPersons(
    roster,
    (Person p) -> p.getGender() == Person.Sex.MALE
        && p.getAge() >= 18
        && p.getAge() <= 25
);
```

Krok 6: Použití existujících funkcionálních rozhraní [3/3]

- Zamysleme se nad rozhraním `Predicate<T>` a jeho použitím ještě jednou:

```
interface Predicate<T> {
    boolean test(T t);
}
...
(Person p) -> p.getGender() == Person.Sex.MALE
    && p.getAge() >= 18
    && p.getAge() <= 25
```

Typování `Predicate<T>`

- Všimněte si, že rozhraní `Predicate` používá typ `T` pouze na zjištění typu vstupního argumentu metody.
- V řadě případů lze typ argumentu odvodit z kontextu pomocí *type inference*, pak je tato

informace nadbytečná, ale nevadí.

Q

Mohli bychom deklaraci typu při volání vynechat?

A

Ano. Následující fragment kódu je rovněž správně. Že `T` odpovídá `Person` zjistí Java při překladu.

```
p -> p.getGender() == Person.Sex.MALE
    && p.getAge() >= 18
    && p.getAge() <= 25
```

Jaká máme funkcionální rozhraní v Javě?

- Rozhraní v `java.util.function` package, například

  Predicate<T>

  s jednou metodou `boolean test(T t)`,

  Supplier<T>

  s jednou metodou `void get(T t)`,

  Consumer<T>

  s jednou metodou `void accept(T t)`.

- Další rozhraní, která sice mohou definovat více metod, ale jen jedna zni je nestatická, například
  - `Comparator<T>` z `java.util`,
  - `Iterable<T>` z `java.lang`.

Příklad použití dalších funkcionálních rozhraní

- Naše současná implementace vyhledávací metody vypisuje informace o osobách splňujících predikát:

```
public static void printPersons(List<Person> roster, Predicate<Person> tester) {
    for (Person p : roster) {
        if (tester.test(p)) {
            p.printPerson();
        }
    }
}
```

- Co když ale s osobami, které splňují predikát daný parametrem `tester`, chceme dělat něco jiného, než jen vypisovat informace o nich?

Funkcionální rozhraní *Consumer*

- Funkcionální rozhraní `Consumer<T>` s metodou `void accept(T t)` nabízí obecnou operaci na zpracování objektu.
- Jako implementaci rozhraní `Consumer<T>` lze použít jakýkoliv kód, který na daném objektu něco vykoná, ale nic nevrací.
- Velmi často se prostě zavolá bezparametrická metoda definovaná na daném objektu, v našem případě `p.printPerson()`.

Příklad definice zpracování s `accept`

```
public static void processPersons(
        List<Person> roster,
        Predicate<Person> tester,
        Consumer<Person> block) {
    for (Person p : roster) {
        if (tester.test(p)) {
            block.accept(p);
        }
    }
}
```

Příklad zpracování

```
processPersons(
     roster,
     p -> p.getGender() == Person.Sex.MALE
         && p.getAge() >= 18
         && p.getAge() <= 25,
     p -> p.printPerson()
);
```

Funkcionální rozhraní *Function*

- Co když nám nestačí zpracovat původní objekty `Person`, ale rádi bychom z nich vytáhli nějaké informace, například e-mail, a teprve tyto informace zpracovali?
- K tomu potřebuje rozhraní, které by vracelo nějakou hodnotu.
- Funkcionální rozhraní `Function<T,R>` nabízí metodu `R apply(T t)`, která slouží k "transformaci" dat typu `T` na data typu `R`.

Příklad definice zpracování s `apply` a `accept`

```
public static void processPersons(
        List<Person> roster,
        Predicate<Person> tester,
        Function<Person, String> mapper,
        Consumer<String> block) {
    for (Person p : roster) {
        if (tester.test(p)) {
            String data = mapper.apply(p);
            block.accept(data);
        }
    }
}
```

Příklad zpracování

```
processPersons(
    roster,
    p -> p.getGender() == Person.Sex.MALE
        && p.getAge() >= 18
        && p.getAge() <= 25,
    p -> p.getEmailAddress(),
    email -> System.out.println(email)
);
```

Extenzivní využití generických typů [1/2]

- Metoda `processPersons` sice pracuje pouze se dvěma typy (`Person` and `String`), ale její smysl se dá zobecnit takto: procházej objekty, vyber, které splňují predikát, vezmi z nich nějaká data, a tato data zpracuj.
- Tohoto zobecnění lze dosáhnout zobecněním typů za použití generik:

```
public static <X, Y> void processElements(
        Iterable<X> source,
        Predicate<X> tester,
        Function <X, Y> mapper,
        Consumer<Y> block) {
    for (X p : source) {
        if (tester.test(p)) {
            Y data = mapper.apply(p);
            block.accept(data);
        }
    }
```

```
}
```

Extenzivní využití generických typů [2/2]

- Vypsání e-mailových adres lidí pak lze vypsat stejně jako předtím:

```
processElements(roster,
    p -> p.getGender() == Person.Sex.MALE
        && p.getAge() >= 18
        && p.getAge() <= 25,
    p -> p.getEmailAddress(),
    email -> System.out.println(email)
);
```

- Co se ale změnilo je to, že lze takto zpracovat i jiné třídy/objekty, než jen `Person`!

Datové proudy (streams)

- Rozhraní `java.util.stream.Stream<T>` používá nastíněné principy a nabízí jednoduché metody, které mohou být použité pro proudové zpracování dat. Náš kód lze přepsat takto:

```
roster.stream()
    .filter(
        p -> p.getGender() == Person.Sex.MALE
            && p.getAge() >= 18
            && p.getAge() <= 25)
    .map(p -> p.getEmailAddress())
    .forEach(email -> System.out.println(email));
```

Shrnutí: Principy proudového zpracování dat

- Java Core API nabízí jednoduchá funkcionální rozhraní vhodná pro proudové zpracování dat.
- Díky generickým typům jsou tato rozhraní jsou nezávislá na tom, jaká data jsou v proudu uložena.
- Rozhraní `Stream` poskytuje metody, které využívají funkcionální rozhraní pro definici jednoduchých operací nad proudovými daty.
- Operace je aplikována na všechny objekty proudu.
- Většina těchto metod vrací "výsledný" proud jako výstupní hodnotu, takže lze operace snadno řetězit.

Shrnutí/2

- Vývojáři mohou využít anonymní třídy pro snadnou definici proudových operací (implementaci požadovaných rozhraní).
- Protože se ale jedná o funkcionální rozhraní, lze navíc použít kompaktní zápis pomocí lambda výrazů.
- Resumé: Vývojáři mohou implementovat proudově-orientované zpracování dat s použitím přístupu podobného funkcionálním jazykům.