Objektovo orientované programovanie

(vnorené typy – 2. časť (anonymné triedy, lambda výrazy))

10. prednáška

Vladislav Novák
FEI STU v Bratislave
18.11.2014

Obsah

Anonymné vnútorné triedy

Existujú dva špeciálne typy vnorených tried. Možno ich deklarovať v rámci tela metódy:
- *lokálne vnútorné triedy* (*local class*) (preberali sme skôr)
- *anonymné vnútorné triedy* (*anonymous class*)

Anonymné triedy sú triedy deklarované v rámci tela metódy, ale nemajú názov. Umožňujú deklarovať triedu a súčasne vytvoriť jej inštanciu. Používajú sa vtedy ak je potrebné použiť iba jednu inštanciu vnútornej triedy.

Pri vytváraní anonymnej triedy uvádzame názov triedy od ktorej bude anonymná trieda dediť, alebo názov rozhrania, ktoré bude anonymná trieda implementovať.

Pri definovaní anonymnej triedy je niekoľko obmedzení, napr. nie je možne deklarovať konštruktor.

Anonymné triedy sú výrazy, to znamená, že môžeme definovať triedu vo vnútri iného výrazu.

V nasledujúcom príklade budú anonymné triedy implementujúce rozhranie a dediace od nadtriedy.

súbor RozhranieUloha.java
```
public interface RozhranieUloha {
    void pracuj();
}
```

súbor AbstraktnaUloha.java
```
public abstract class AbstraktnaUloha {
    private int cislo;

    public AbstraktnaUloha(int cislo) {
        this.cislo = cislo;
    }

    public abstract void pracuj();

    protected int dajCislo() {
        return cislo;
    }
                                                                    }
```

súbor SpravcaUloh1.java
```
public class SpravcaUloh1 {
    private RozhranieUloha uloha;

    public void nastavUlohu(RozhranieUloha uloha) {
        this.uloha = uloha;
    }

    public void spustiUlohu() {
        uloha.pracuj();
    }
}
```

súbor SpravcaUloh2.java

```
public class SpravcaUloh2 {
    private AbstraktnaUloha uloha;

    public void nastavUlohu(AbstraktnaUloha uloha) {
        this.uloha = uloha;
    }

    public void spustiUlohu() {
        uloha.pracuj();
    }
}
```

súbor PrikladPreAnonimneTriedy.java

```
public class PrikladPreAnonimneTriedy {
    public static void main(String[] args) {
        SpravcaUloh1 spravcaUloh1a = new SpravcaUloh1();
        SpravcaUloh2 spravcaUloh2a = new SpravcaUloh2();
        SpravcaUloh1 spravcaUloh1b = new SpravcaUloh1();
        SpravcaUloh2 spravcaUloh2b = new SpravcaUloh2();

        RozhranieUloha uloha1 = new RozhranieUloha() {
            @Override
            public void pracuj() {
                System.out.println("uloha1a");
            }
        };

        AbstraktnaUloha uloha2 = new AbstraktnaUloha(10) {
            @Override
            public void pracuj() {
                System.out.println("uloha2a (" + dajCislo() + ")");
            }
        };

        spravcaUloh1a.nastavUlohu(uloha1);
        spravcaUloh2a.nastavUlohu(uloha2);

        spravcaUloh1b.nastavUlohu(new RozhranieUloha() {
            @Override
            public void pracuj() {
                System.out.println("uloha1b");
            }
        });

        spravcaUloh2b.nastavUlohu(new AbstraktnaUloha(10) {
            @Override
            public void pracuj() {
                System.out.println("uloha2b (" + dajCislo() + ")");
            }
        });

        //niekde d'alej
        spravcaUloh1a.spustiUlohu();
        spravcaUloh2a.spustiUlohu();
        spravcaUloh1b.spustiUlohu();
        spravcaUloh2b.spustiUlohu();
    }
}
```

Lambda výrazy

Anonymné triedy umožňujú kratší zápis kódu. Napriek tomu, najmä v situáciách kedy trieda obsahuje iba jednu jednoduchú metódu, je použitie anonymných tried zbytočne neprehľadne a tiež dlhé.

Lambda výrazy umožňujú v jave pomocou stručného zápisu definovať triedy s jednou metódou a zároveň vytvárať ich inštancie. Keďže sa vytvára iba jedna inštancia danej triedy, ktorá má iba jednu metódu, názov triedy a metódy neuvádzame. Uvádzajú sa len argumenty a definícia metódy.
Lambda výrazy sa používajú v prípadoch keď je argumentom metódy ďalšia funkcionalita, ktorá bude v metóde vykonávaná.

Funkcionálne rozhranie (*functional interface*) je rozhranie, ktoré obsahuje práve jednu abstraktnú metódu. Inštancie implementujúce funkcionálne rozhranie možno definovať pomocou lambda výrazov.

Nech je definovaný enumeračný typ `TypStudia` a trieda `Student`

```
public enum TypStudia {
    BAKALARSKE, INZINIERSKE
}
```

```
public class Student {
    private String meno;
    private TypStudia typStudia;
    private double studijnyPriemer;

    public Student(String meno, TypStudia typStudia,
                                          double studijnyPriemer) {
        this.meno = meno;
        this.typStudia = typStudia;
        this.studijnyPriemer = studijnyPriemer;
    }

    public String dajMeno() {
        return meno;
    }

    public TypStudia dajTypStudia() {
        return typStudia;
    }

    public double dajStudyjnyPriemer() {
        return studijnyPriemer;
    }

    public void nastavTypStudia(TypStudia typStudia) {
        this.typStudia = typStudia;
    }

    public void nastavStudyjnyPriemer(double studyjnyPriemer) {
        this.studijnyPriemer = studyjnyPriemer;
    }
```

```
    @Override
    public String toString() {
        return "meno=" + meno + ", typStudia=" + typStudia
                        + ", studyjnyPriemer=" + studijnyPriemer;
    }
}
```

Nech je v metóde `main` definované pole študentov

```
Student[] studenti = {
    new Student("Jano",  TypStudia.BAKALARSKE,  1),
    new Student("Peter", TypStudia.BAKALARSKE,  2),
    new Student("Tibor", TypStudia.BAKALARSKE,  1.5),
    new Student("Roman", TypStudia.INZINIERSKE, 1.4),
    new Student("Juraj", TypStudia.INZINIERSKE, 2.5)
};
```

Definujme metódu, ktorá vypíše všetkých študentov v poli, spĺňajúcich určitú podmienku. Typ podmienky môže byť ľubovoľný, preto bude parametrom metódy objekt, obsahujúci metódu, ktorá vyhodnotí podmienku. Metóda vyhodnocujúca podmienku je definovaná funkcionálnym rozhraním

```
public interface Podmienka {
    boolean vyhodnot(Student student);
}
```

Definícia metódy

```
public static void vypis(Student[] studenti, Podmienka podmienka) {
    for(Student student : studenti) {
        if( podmienka.vyhodnot(student) ) {
            System.out.println(student);
        }
    }
    System.out.println();
}
```

Teraz vypíšeme všetkých študentov, ktorý sú na bakalárskom štúdiu. Použijeme anonymnú triedu

```
Podmienka podmienka1 = new Podmienka() {
    @Override
    public boolean vyhodnot(Student student) {
        return student.dajTypStudia() == TypStudia.BAKALARSKE;
    }
};

vypis(studenti, podmienka1);
```

alebo kratšie

```
vypis(studenti, new Podmienka() {
    @Override
    public boolean vyhodnot(Student student) {
        return student.dajTypStudia() == TypStudia.BAKALARSKE;
    }
});
```

Keďže objekt testujúci typ štúdia má iba jednu metódu, môžeme použiť lambda výrazy.

Lambda výraz je podobný anonymnej metóde. Príklad syntaxe:

```
(Typ1 parameter1, Typ parameter2) -> { príkaz1; príkaz2; }
```

Lambda výraz sa skladá zo

- zoznamu parametrov uvedených v zátvorkách, oddelených čiarkou. Typ parametrov môžeme vynechať. Ak má lambda výraz práve jeden parameter, môžeme vynechať zátvorky.
- šípka `->`
- implementácia definovaná blokom príkazov, alebo jedným výrazom, alebo jedným príkazom
    - Ak je implementácia definovaná v bloku príkazov, tak je blok príkazov uzavretý v zložených zátvorkách. Príkazy sú oddelené bodkočiarkou.
    - Ak je implementácia definovaná jedným výrazom, tak sa zložené zátvorky nemusia uvádzať. V tomto prípade bude návratovou hodnotou lambda výrazu hodnota uvedeného výrazu
    - Ak lambda výraz nemá návratovú hodnotu (vracia typ `void`) a implementáciou je jediný príkaz, môžeme v niektorých prípadoch vynechať zložené zátvorky (potom musíme vynechať aj bodkočiarku).

Typy parametrov a návratový typ lambda výrazu sa určia podľa kontextu. Lambda výrazy môžeme použiť iba v prípadoch, keď kompilátor dokáže určiť typy parametrov a návratovej hodnoty.

Príklad použitia lambda výrazu pre výpis študentov bakalárskeho štúdia

```
Podmienka podmienka2 =
    (Student student) ->
        {return student.dajTypStudia() == TypStudia.BAKALARSKE; };
vypis(studenti, podmienka2);
```

Môžeme použiť aj kratší zápis.

```
Podmienka podmienka3
        = student -> student.dajTypStudia() == TypStudia.BAKALARSKE;
vypis(studenti, podmienka3);
```

V tomto zápise sme vynechali typ parametra, zátvorky ohraničujúce parametre. V definícii implementácie sme namiesto príkazu `return` s hodnotou použili jednoduchý výraz, preto sme vynechali zložené zátvorky.

Lambda výraz môžeme definovať priamo pri volaní metódy

```
vypis(studenti, (Student student) ->
    {return student.dajTypStudia() == TypStudia.BAKALARSKE; }
);
```

alebo kratšie

```
vypis(studenti,
         student -> student.dajTypStudia() == TypStudia.BAKALARSKE);
```

Prístup ku premenným existujúcim v okolí definovania anonymnej triedy, alebo lambda výrazu

Anonymné triedy a lambda výrazy sú lexikálne súčasťou kódu v ktorom sú použité. Nemôžu definovať nové premenné s rovnakým názvom ako je názov premennej, ktorá existuje v mieste definície anonymnej triedy, alebo lambda výrazu. Môžu tieto existujúce premenné použiť ak sú final, alebo „effectively final“. Nasleduje niekoľko príkladov s popisom.

```
public void metoda() {
    final int b = 1;
    Predicate<Integer> predicate = (a) -> a > b;
}
```

V lambda výraze môžeme premennú `b` použiť. Premenná `b` musí byť konštantná, čo znamená že musí byť final, alebo „effectively final“. V tomto prípade je premenná `b` final.

```
public void metoda() {
    int b = 1;
    Predicate<Integer> predicate = (a) -> a > b;
}
```

V tomto prípade nie je premenná `b` final, ale je „effectively final“, takže ju môžeme v lambda výraze použiť.

```
public void metoda() {
    int b = 1;
    b++;
  //Predicate<Integer> predicate = (a) -> a > b; //CHYBA kompilácie
}
```

V tomto prípade nie je premenná `b` ani „effectively final“, takže ju nemôžeme použiť v lambda výraze.

```
public void metoda() {
    int a = 1;
  //Predicate<Integer> predicate = (a) -> a > 5; //CHYBA kompilácie
}
```

V tomto prípade kompilátor vyhlási chybu, pretože premenná `a` je už deklarovaná (pred definíciou lambda výrazu).

Požitie štandardných funkcionálnych rozhraní

Rozhranie `Podmienka` je funkcionálnym rozhraním, ktoré sme definovali spolu s metódou `vypis(Student[], Podmienka)`. Aby sme nemuseli definovať rozhrania spolu s metódami, Java obsahuje viacero štandardných funkcionálnych rozhraní, ktoré môžeme použiť. Tieto rozhrania sú definované v balíku `java.util.function`.

V tomto balíku je napr. definované rozhranie `Predicate<T>`, ktoré obsahuje metódu s jedným parametrom vracajúcu `boolean`, tak ako naša podmienka.

```
public interface Predicate<T> {
    boolean test(T t);
    //ďalšie default-né a statické metódy
}
```

Pri definícii metódy môžeme namiesto nášho rozhrania, použiť toto štandardné rozhranie. Aj po zmene použitého rozhrania bude použitie metódy bude rovnaké. Názvy metód v rozhraniach `Podmienka` a `Predicate<T>` sú odlišné, ale pri použití lambda výrazov, názov metódy neuvádzame.

Definícia metódy vypisujúcej študentov spĺňajúcich podmienku zadanú ako parameter

```
public static void vypis2(Student[] studenti,
                                    Predicate<Student> podmienka) {
    for(Student student : studenti) {
        if( podmienka.test(student) ) {
            System.out.println(student);
        }
    }
    System.out.println();
}
```

Volanie metódy môže byť rovnaké ako predtým

```
vypis2(studenti,
         student -> student.dajTypStudia() == TypStudia.BAKALARSKE);
```

Teraz zovšeobecnime metódu tak, aby sme pre vybraných študentov mohli vykonať ľubovoľnú akciu. Nato využijeme ďalšie štandardné rozhranie

```
public interface Consumer<T> {
    void accept(T t)
    //ďalšia default-ná metóda
}
```

Definícia zovšeobecnenej metódy

```
public static void metoda3(Student[] studenti,
                           Predicate<Student> podmienka,
                           Consumer<Student> vykonajPreVybaneho) {
    for(Student student : studenti) {
        if( podmienka.test(student) ) {
            vykonajPreVybaneho.accept(student);
        }
    }
    System.out.println();
}
```

Volanie metódy

```
metoda3(studenti,
    student -> student.dajTypStudia() == TypStudia.BAKALARSKE,
    student -> System.out.println(student) );
```

Vytvorme metódu pomocou ktorej vypočítame priemerný študijný priemer všetkých vybraných študentov. Definujeme ju tak, aby mohla vypočítať priemer ľubovoľných číselných hodnôt súvisiacich s vybranými študentmi. Využijeme rozhranie `Function<T,R>`

```
public interface Function<T,R> {
    R apply(T t);
    //ďalšie default-né a statické metódy
}
```

Definícia metódy (využijeme obálkovú triedu a automatické rozbalovanie)

```
public static double vypocitajPriemer (Student[] studenti,
                                Function<Student, Double> funkcia) {
    double suma = 0;
    for(Student student : studenti) {
        suma += funkcia.apply(student);
    }
    return suma / studenti.length;
}
```

Volanie metódy

```
double priemer = vypocitajPriemer (studenti,
                           student -> student.dajStudyjnyPriemer());
```

Použitie referencii na metódy

V niektorých prípadoch lambda výraz vykoná iba volanie existujúcej metódy. Vtedy namiesto lambda výrazu volajúceho existujúcu metódu, môžeme použiť priamo referenciu na túto metódu.

Typy referencii na metódy:

- referencia na statickú metódu
- referencia na konštruktor
- referencia na inštančnú metódu definovaného objektu (určíme aj objekt nad ktorým sa metóda volá)
- referencia na inštančnú metódu ľubovoľného objektu definovaného typu

Referencia na statickú metódu

Budeme plniť pole hodnotami, ktoré budú generované statickou metódou. Využijeme štandardné funkcionálne rozhranie

```
public interface Supplier<T> {
    T get();
}
```

Definícia metódy

```
public static void naplnPole(double[] pole,
                                        Supplier<Double> vyrobca) {
    for(int i = 0; i < pole.length; i ++) {
        pole[i] = vyrobca.get();
    }
}
```

Volanie metódy (lambda výraz iba zavolá metódu `Math.random()`)

```
double[] pole1 = new double[5];
naplnPole(pole1, () -> Math.random() );
```

Volanie metódy s využitím referencie na metódu `Math.random()`

```
double[] pole1 = new double[5];
naplnPole(pole1, Math::random);
```

Definujme metódu tak, aby bola použiteľná pre polia ľubovoľného referenčného typu (metóda bude generická). Definovanie generickej metódy nesúvisí s referenciami na metódy, ale ďalej budeme definovať generické metódy.

```
public static <T> void naplnPole(T[] pole, Supplier<T> vyrobca) {
    for(int i = 0; i < pole.length; i ++) {
        pole[i] = vyrobca.get();
    }
}
```

Volanie tejto metódy

```
Double[] pole2 = new Double[5];
naplnPole(pole2, Math::random);
```

Referencia na konštruktor

```
public static <ELEMENT, VALUE> void naplnPole(ELEMENT[] pole,
                                Function<VALUE, ELEMENT> konvertor,
                                Supplier<VALUE> generator) {
    for(int i = 0; i < pole.length; i ++) {
        VALUE value = generator.get(); //vygenerovanie parametra pre konštruktor
        pole[i] = konvertor.apply(value); //zavolanie konštruktora
    }
}
```

Volanie metódy

```
Double[] pole3 = new Double[5];
naplnPole(pole3, Double::new, Math::random);
```

Za parameter metódy `generator`, je pri volaní doplnená referencia na metódu generujúcu náhodné čísla. Za parameter metódy `konvertor`, je pri volaní doplnená referencia na konštruktor triedy `Double`, ktorého parametrom bude hodnota vygenerovaná metódou `Math.random()`.

Referencia na inštančnú metódu definovaného objektu

Definujme metódu, ktorá bude tlačiť tabuľku údajov v poli. Každý riadok tabuľky bude obsahovať informácie o jednom prvku pola. Vhodné naformátovanie tabuľky bude zabezpečovať `formatovac`, ktorý je vstupným parametrom metódy. Pre každý prvok poľa vytvorí text, ktorý bude vytlačený do riadku.

```
public static <T> void vytlacTabulku(T[] pole,
                                Function<T, String> formatovac ) {
    for(T prvok : pole) {
        String riadok = formatovac.apply(prvok);
        System.out.println(riadok);
    }
}
```

Definícia formátovača pre tabuľku študentov

```
public class Formatovac {
    private int sirkaStlpca1;
    private int sirkaStlpca2;

    public FormatovacRiadkovTabulkyStudentov(int sirkaStlpca1,
                                             int sirkaStlpca2) {
        this.sirkaStlpca1 = sirkaStlpca1;
        this.sirkaStlpca2 = sirkaStlpca2;
    }

    public String vytvorRiadok(Student student) {
        return "|"
            + prisposobRetazec(sirkaStlpca1, student.dajMeno())
            + "|"
            + prisposobRetazec(sirkaStlpca2, typAkoString(student))
            + "|";
    }

    private String prisposobRetazec(int sirka, String retazec) {
        char[] pismena = new char[sirka];
        for(int i = 0; i < sirka && i < retazec.length(); i ++) {
            pismena[i] = retazec.charAt(i);
        }
        for (int i = retazec.length(); i < pismena.length; i++) {
            pismena[i] = ' ';
        }
        return String.valueOf(pismena);
    }

    private String typAkoString(Student student) {
        switch(student.dajTypStudia()) {
            case BAKALARSKE:
                return "bc.  studium";
            case INZINIERSKE:
                return "ing. studium";
            default:
                throw new RuntimeException("chybne impl. metoda");
        }
    }
}
```

Volanie metódy bez využitia referencie na metódu

```
Formatovac formatovac = new Formatovac(5, 12);
vytlacTabulku(studenti, student ->formatovac.vytvorRiadok(student));
```

Volanie metódy s využitím referencie na metodu

```
Formatovac formatovac = new Formatovac(5, 12);
vytlacTabulku(studenti, formatovac::vytvorRiadok);
```

Referencia na inštančnú metódu ľubovoľného objektu definovaného typu

Vyššie uvedenú metódu
`vypocitajPriemer (Student[], Function<Student, Double>)`
sme volali takto

```
double priemer = vypocitajPriemer (studenti,
                                   student -> student.dajStudyjnyPriemer());
```

Lambda výraz pre zadanú inštanciu študenta zavolá metódu vracajúcu jeho študijný priemer. Namiesto tohto zápisu môžeme využiťreferenciu na inštančnú metódu.

```
double studijnyPriemer = vypocitajPriemer(studenti,
                                          Student::dajStudyjnyPriemer);
```

Prúdy údajov

Prvky poľa môžeme spracovávať vďaka lambda výrazom oveľa jednoduchšie. Môžeme využiť prúd údajov, ktorý obsahuje prvky pola.

Príklad výpisu všetkých študentov bakalárskeho štúdia

```
Arrays.stream(studenti)
    .filter(student -> student.dajTypStudia()==TypStudia.BAKALARSKE)
    .forEach(student -> System.out.println(student));
```

Príklad získania priemerného študijného priemeru

```
double priemer = Arrays.stream(studenti)
    .mapToDouble(student -> student.dajStudyjnyPriemer())
    .average()
    .getAsDouble();
```