Reflexe 11

V tomto semináři se seznámíme s reflexí (Reflection API), rozhraním, které umožňuje pokročilou práci s objekty a třídami, která není možná pomocí běžných prostředků jazyka Java.

1 Práce s vlastnostmi tříd

Každá třída, označme si ji `T`, má vyčleněný speciální atribut `class`, který obsahuje odkaz na objekt třídy `Class<T>`, pomocí nějž můžeme podrobně zkoumat vlastnosti této třídy a jejich instancí. Objekt této třídy se dá taktéž získat zavoláním metody `Object.getClass()`.

Možnosti reflexe si budeme demonstrovat na jednoduché třídě popisující zaměstnance.

```
public class Employee {
    public static final int MIN_AGE = 15;
    private String name;
    private int age;
    private double salary;

    public Employee() {
        super();
    }
    public Employee(String name, int age, double salary) {
        this.name = name;
        this.age = age;
        this.salary = salary;
        if (age < MIN_AGE)
            throw new IllegalArgumentException("Child labor is prohibited");
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
```

```
    public void setAge(int age) {
        this.age = age;
    }
    public double getSalary() {
        return salary;
    }
    public void setSalary(double salary) {
        this.salary = salary;
    }
    @Override
    public String toString() {
        return "Employee [name=" + name + ", age=" + age + ", salary=" + salary + "]";
    }
}
```

Nejdříve získáme objekt popisující naši ukázkovou třídu.

```
Class<Employee> clazz = Employee.class;
```

Typový parametr udává, jakou třídu objekt třídy `Class` popisuje. Pokud to nevíme, můžeme tento parametr nechat volný, tj. použít `Class<?>`. Běžná konvence by velila proměnnou obsahující objekt třídy `Class<T>` pojmenovat `class`. To je ale klíčové slovo, které takto nelze použít, a proto se obvykle jako název takové proměnné používá `clazz` nebo `klass`.

S tímto objektem můžeme nyní pracovat a například zjistit, jaké atributy nebo metody třída má. Následující kód přečte seznam atributů.

```
Field[] fields = clazz.getDeclaredFields();
for (Field field : fields) {
    System.out.println(
        String.format("%s: %s (%s)",
            field.getName(),
            field.getType(),
            Modifier.toString(field.getModifiers())));
}
```

Na výstupu bychom měli získat:

```
MIN_AGE: int (public static final)
name: class java.lang.String (private)
age: int (private)
salary: double (private)
```

Poznamenejme, že metoda `Field.getType()` vrací opět instance třídy `Class<T>`, a je tak možné podrobně zkoumat dané atributy. Metoda `Field.getModifiers()` vrací informace o použitých modifikátorech atributů v podobě číselné hodnoty. Tu lze dále analyzovat pomocí pomocné třídy `Modifier`, kterou jsme použili na převod modifikátoru do textového popisu.

Podobně můžeme získat seznam metod.

```
Method[] methods = clazz.getDeclaredMethods();
for (Method method : methods) {
    System.out.println(method.getName());
}
```

Všimněme si, že tento seznam není kompletní a některé metody chybí, např. `Object.hashCode()`, `Object.equals(Object)`. Je to proto, že metoda `Class.getDeclaredMethods`, podobně jako metoda `Class.getDeclaredFields`, vrací pouze informace o metodách a atributech, které jsou deklarováné v dané třídě. Pokud chceme získat výčet kompletní, musíme použít `Class.getMethods()` nebo `Class.getFields()`.

V podobném duchu lze zkoumat další vlastnosti, např. jaký je předek dané třídy, jaké implementuje rozhraní a podobně.

V případě platformy Java instance třídy `Class<T>` není objekt, který by definoval třídu `T`, k čemuž by to mohlo svádět a z pohledu elegance jazyka by to jistě dávalo smysl. Instance třídy `Class<T>` bychom měli chápat pouze jako prostředek (rozhraní) ke zkoumání jednotlivých tříd a objektů daného programu. Skutečná podstata tříd a objektů (kód vykonávaný CPU, skutečné uložení dat) nám zůstává (a měla by zůstat) utajena.

2 Manipulace s objekty pomocí reflexe

Reflexe umožňuje nejen zkoumat třídy, jejich atributy a metody, ale taktéž k nim přistupovat, tzn. číst a měnit obsah atributů nebo volat metody[1], ale taktéž vytvářet jejich instance.

Máme-li v proměnné `clazz` objekt třídy `Class` popisující třídu `Employee` můžeme sestrojit instance této třídy, jak ukazuje tento kód.

```
Constructor<?> constructor = clazz.getConstructor();
Object obj = constructor.newInstance();
```

Nejdříve získáme objekt konstruktoru, a následně pomocí něj vytvoříme instanci dané třídy. Že se jedná skutečně o objekt třídy Employee, si můžete ověřit například zavoláním metody Object.toString().

V podobném duchu můžeme pracovat s metodami. Nejdříve získáme objekt odpovídající dané metodě:

```
Method setNameMethod = clazz.getMethod("setName", String.class);
```

První argument určuje jméno metody a následuje výčet typů jednotlivých argumentů.[2] Následně můžeme metodu zavolat.

```
setNameMethod.invoke(obj, "Sova Antonin");
```

[1] A to včetně těch soukromých!

[2] Nelze použít pouze jméno, protože jména metod mohou být přetížena, tj. mohou existovat dvě metody stejného jména lišící se v parametrech.

První argument metody `invoke` představuje objekt, jehož metodu budeme volat, zbývající argumenty jsou argumenty, které budou předány volané metodě.[3]

Vidíme, že reflexe nám umožňuje manipulovat s objekty mimo prostředky samotného jazyka Java. Všimněme si, že dosud, když jsme vytvářeli objekt, museli jsme použít operátor `new` a to jaký objekt bude vytvořen bylo rozhodnotu již v době překladu. To nyní již neplatí, protože jsme potenciálně schopni vytvořit libovolnou instanci libovolné třídy, pro kterou máme odpovídající objekt třídy `Class`[4]. Podobně je to s voláním metod pomocí operátoru . (tečka), kdy je volaná metoda určena čistě na základě kódu programu. S reflexí to již neplatí a volanou metodu můžeme určit pomocí proměnné typu `Method`. Díky tomu je možné pracovat shodně s nepříbuznými objekty. Uvažujme například třídu `Animal` s podobnými metodami, jako má třída `Employee`.

```
public class Animal {
    private String name;
    private int age;

    public Animal() {
        super();
    }
    public Animal(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
    @Override
    public String toString() {
        return "Animal [name=" + name + ", age=" + age + "]";
    }
}
```

Můžeme s ní pracovat stejnými operacemi jako s třídou `Employee`, vyzkoušejte si:

[3] Voláme-li statickou metodu, použijeme `null` místo objektu.

[4] Od Javy 9 existují omezení, která nám v tom dokáží zabránit.

```
Class<?> clazz = Animal.class;

Constructor<?> constructor = clazz.getConstructor();
Object obj = constructor.newInstance();
Method setNameMethod = clazz.getMethod("setName", String.class);
setNameMethod.invoke(obj, "Sova Antonin");
// obj => Animal [name=Sova Antonin, age=0]
```

Zároveň je potřeba dodat, že reflexe sice umožňuje věci, které v jazyce Java nelze realizovat, je to však za cenu dopadu na rychlost programu. Reflexe je pomalá (překladač nemůže optimalizovat jednotlivá volání, kontroly hodnot, atp.), nabourová očekávanou strukturu kódu (IDE nemůže korektně udělat refactoring nebo hlásit chyby), a proto by se s ní mělo šetřit.

Reflexe sice přináší do Javy dynamiku, kterou můžeme znát z jiných jazyků (např. Lisp), ale měli bychom mít stále na paměti, že Java není dynamicky typovaný jazyk, a tak s ním i pracovat. Proto, pokud se v kódu začne nad míru objevovat použití reflexe, je to pravděpodobně příznak, že je návrh udělán špatně a měl by být přehodnocen. Např. místo dynamického vytváření objektů pomocí reflexe, lze často použít návrhový vzor *Factory*, místo volání metod pomocí reflexe lze použít vhodně navržená rozhraní, atd.

3 Anotace

Zajímavým prvkem jazyka Java, se kterými jsme se již okrajově setkávali a který ve spojení s reflexí představuje mocnou zbraň, jsou *anotace*.

Anotace jsou element jazyka Java, pomocí nejž můžeme doplnit informaci ke třídám, metodám, atributům, argumentům, typům, anotacím, apod. S touto informací může pracovat překladač, další nástroj nebo program sám pomocí reflexe. Už jsme se s nimi nejednou setkali, např. pokud metoda překrývá metodu předka, je tato metoda označena anotací `@Override`.

Touto anotací programátor deklaruje, že by metoda měla překrývat metodu v předkovi. Například v následující situaci nás překladač upozorní na chybu.

```
public class Foo {
   public int inc(int i) {
      return i + 1;
   }
}

public class Bar extends Foo {
    @Override
    public double inc(double i) {
       return i + 2;
    }
}
```

Pokud by nás překladač na tento typ chyby neupozornil, měli bychom v programu chybu, která se může

docela umně skrývat.

S další anotací, se kterou jsme se asi již setkali, je anotace `@Deprecated`, která slouží k označení metod a atributů, které jsou zastaralé a neměly by se proto používat. Tuto informaci například využívají IDE a zobrazují ji v kódu.

K čemu se anotace nakonec použijí záleží čistě na potřebách programátora, protože Java umožňuje deklarovat vlastní anotace.

3.1 Práce s anotacemi

Přístup k anotacím je realizován přes reflexi. Použití si ukážeme na anotaci `@Deprecated` a naprosto neužitečné třídě.

```
@Deprecated
public class Useless {
    @Deprecated(since = "1.2")
    public void nothingToDo() {
    }

    @Override
    public String toString() {
        return "useless class";
    }
}
```

Jeden způsob, jak zjistit anotace, je vyžádat si všechny anotace pomocí metody `Class.getAnnotations()` vracející pole anotací. (Všechny anotace jsou potomky třídy `Annotation`.)

```
Annotation[] annots = clazz.getAnnotations();
```

A pak se podívat, zda anotace, se kterou chceme pracovat, je tomto v poli, nebo ne.

```
for (Annotation a : annots) {
    if (a instanceof Deprecated)
        System.out.println("class is deprecated");
}
```

Záměrně uvádíme *pracovat*, protože anotace nemusí sloužit pouze k vyznačení nějaké vlastnosti, ale mohou nést další informaci, jak lze vidět například metody `Useless.nothingToDo()`, ze které můžeme vyčíst, odkdy je metoda označena jako `@Deprecated`, jak ukazuje následující kód.

```
Method m = clazz.getMethod("nothingToDo");
Deprecated deprAnn = m.getAnnotation(Deprecated.class);
System.out.println("Deprecated since: " + deprAnn.since());
```

V tomto příkladu jsme si vyžádali konkrétní anotaci typu `Deprecated` a získali jsme ji jako objekt, který má další metody, které poskytují další informace, jež byly uvedeny společně s touto anotací, např. `Deprecated.since()`.

Variantu tohoto příkladu ještě vyzkoušíme s metodou `Useless.toString()` a anotací `@Override`.

```
Method m = clazz.getMethod("toString");
Override overrideAnn = m.getAnnotation(Override.class);
// overrideAnn => null
```

V tomto případě metoda `getAnnotation` vrátí `null`, přestože metoda `toString()` opravdu má danou anotaci. Je to dáno tím, že jednoltivé anotace mají určené, kdy mají být k dispozici – (i) ve fázi překladu, (ii) v přeloženém souboru, (iii) v běžícím programu. Zatímco anotace `@Deprecated` spadá do kategorie (iii), anotace `@Override` je k dispozici jen v době překladu (i), a proto ji nemůžeme přečíst za běhu pomocí reflexe.

3.2 Tvorba vlastních anotací

Jak jsme již naznačili, množina anotací není omezená a programátor si může vytvořit vlastní a používat je podle vlastních potřeb.

My si tuto funkcionalitu ukážeme na metodě, která nám na zakládě anotace vytáhné informace z objektu a zobrazí je jako zformátovaný řetězec.

Nejdříve si vytvoříme anotaci `@Inspect`, kterou budeme označovat metody a atributy, které budeme chtít mít ve výpisu:

```
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

import java.lang.annotation.Retention;
import java.lang.annotation.Target;

@Retention(RUNTIME)
@Target({ FIELD, METHOD })
public @interface Inspect {
}
```

Anotace se deklaruje pomocí klíčového slova `@interface` (zavináč + `interface`). Dále jsou u anotace uvedeny dvě anotace: (i) `@Retention` udávající, kdy má být anotace k dispozici (v našem případě za běhu programu) a (ii) `@Target` která určuje, ke kterým prvkům jazyka je možné anotaci připojit (v našem případě k atributům a metodám).

Nyní si vytvoříme metodu `static String inspectObject(Object obj)`, která se bude starat o přečtení hodnot a jejich zformátování.

```
public static String inspectObject(Object obj) {
    Class<?> clazz = obj.getClass();
    Map<String, String> values = new TreeMap<>();
```

Získáme instanci třídy `Class` objektu, který chceme číst, a vytvoříme si pomocný slovník[5], kam budeme ukládat přečtené hodnoty.

Nejdříve přečteme atributy:

```
for (Field field : clazz.getDeclaredFields()) {
    field.setAccessible(true);
    Inspect ann = field.getAnnotation(Inspect.class);
    if (ann != null) {
        // atribut ma anotaci, a proto jej precteme
        Object value = field.get(obj);
        values.put(field.getName(), value.toString());
    }
}
```

Metodu `Field.setAccessible(true)` voláme, abychom zpřístupnili i soukromé (či jinak nepřístupné) atributy.

Podobný kód pak vytvoříme pro metody:

```
for (Method method : clazz.getDeclaredMethods()) {
    method.setAccessible(true);
    Inspect ann = m.getAnnotation(Inspect.class);
    if (ann != null) {
        // metoda ma anotaci, a proto ji zavolame
        Object value = method.invoke(obj);
        values.put(method.getName(), value.toString());
    }
}
```

Zbývá hodnoty jen zformátovat a hezky vypsat. Zde si vypomůžeme streamy, které nám zformátují páry klíč-hodnota a výsledek spojí do jednolitého řetězce.

```
String valueStr = values.entrySet().stream()
   .map(e -> e.getKey() + " = " + e.getValue())
   .collect(Collectors.joining(", " , "[", "]"));
   return clazz.getName() + " " + valueStr;
```

Nově vytvořenou metodu vyzkoušíme na následující třídě:

[5] `TreeMap`, aby byl výsledek pěkně seřazen.

```
public class Rectangle {
    @Inspect private final double x;
    @Inspect private final double y;
    private final double height;
    private final double width;

    public Rectangle(double x, double y, double height, double width) {
        this.x = x;
        this.y = y;
        this.height = height;
        this.width = width;
    }
    public double getX() {
        return x;
    }
    public double getY() {
        return y;
    }
    @Inspect
    public double getArea() {
        return height * width;
    }
}
```

Zavolání metody `inspectObject(Object)` nám vrátí informace u nichž máme uvedenou anotaci.

```
inspectObject(new Rectangle(10, 20, 30, 40)));
// => cz.upol.zp4jv.lecture11.Rectangle [getArea = 1200.0, x = 10.0, y = 20.0]
```

Náš příklad bychom mohli vyšperkovat tím, že u anotace `@Inspect` budeme moci uvést jméno, pod kterým se má hodnota ve výstupu objevit.

Rozšíříme proto anotaci:

```
public @interface Inspect {
   String as() default "";
}
```

Do anotace jsme přidali signaturu metody, pod kterou bude informace přístupná. Při použití anotace budeme uvádět název „metody" + = hodnota, např.:

```
@Inspect(as = "area")
public double getArea() {
...
}
```

Pokud bychom do signatury metody neuvedli implicitní hodnotu, stala by se hodnota „as" povinnou, což v tomto případě není žádoucí.

Odpovídajícím způsobem ještě upravíme metodu `inspectObject`. Tj. pokud není uvedeno jméno, použijeme název metody nebo atributu, jinak uvedeme jako jméno požadovaný řetězec z anotace.

```
String name = ann.as().isEmpty() ? method.getName() : ann.as();
values.put(name, value.toString());
```