Abstraktní třídy, výčtové typy, záznamy a omezení dědičnosti 6

1 Abstraktní třídy a metody

Dědičnost nám umožňuje modelovat vztahy mezi třídami tak, že z obecnějších tříd jsme schopni odvodit třídy konkrétnější. V minulém semináři jsme například měli vztah *dopravní prostředek* (třída `Vehicle`), *auto* (třída `Car`), *taxi* (třída `Taxi`). Je evidetní, že *dopravní prostředek* je obecnější pojem než *auto*, protože všechna auta jsou podmnožinou dopravních prostředků. Podobně *Taxi* jsou podmnožinou *aut*, resp. *dopravních prostředků*. Může se stát, že některá třída modeluje pojem tak obecný, že není možné vyjádřit kódem všechny její metody a nemá proto smysl vytvářet instance takové třídy. Avšak taková třída není bez užitku, protože můžeme pracovat s potomky této třídy a do obecné třídy vložit funkcionalitu těmto potomkům společnou.

Ukažme si to na příkladu. Uvažujme, že budeme modelovat *celočíselné aritmetické výrazy*. Za celočíselný aritmetický výraz budeme považovat:

- celé číslo (konstanta)
- $a + b$,
- $a - b$,
- $a * b$,
- a / b, kde a a b jsou celočíselné aritmetické výrazy.

U celočíselných aritmetických výrazů můžeme uvažovat metodu `int evaluate()`, která vrátí hodnotu výrazu po jeho vyhodnocení.

Kdybychom si zavedli obecnou třídu `Expression` s touto metodou, dostáváme se do problému, jak určit, co by měla tato metoda dělat a jakou hodnotu vracet. Mohla by třeba vracet 0, -1, nebo jinou, arbitrárně zvolenou, hodnotu.

```
public class Expression {
  public int evaluate() {
    return 0; // ??? je to správně!?
  }
}
```

Mohli bychom si vytvořit i instanci této třídy.

```
Expression e = new Expression();
e.evaluate(); // ==> 0
```

Takto pojatá třída ale nedává koncepčně smysl. Jednak neodpovídá tomu, co je celočíselný aritmetický výraz (viz výčet výše), a navíc třída `Expression` nefunguje jako zobecnění, protože metoda `evaluate()` vrací vždy 0, i když konkrétní třídy budou vracet konkrétní hodnoty podle svých atributů.

Pro řešení takových situací nabízí jazyk Java tzv. *abstraktní metody* a *abstraktní třídy*. Abstraktní metody jsou vyznačeny klíčovým slovem `abstract` a nemají uvedeny tělo. Tělo metody je následně definováno v potomkovi. Třída, jež obsahuje alespoň jednu abstraktní metodu, se označuje jako abstraktní, musí být označena taktéž klíčovým slovem `abstract` a není možné vytvořit její instance.[1]

Třídu `Expression` bychom tedy upravili následovně a rovnou si ukážeme i vytvoření jejího potomka, reprezentující konstantní výraz.

```
public abstract class Expression {
  public abstract int evaluate();
}

public class ConstantExpression extends Expression {
  private final int value;
  public ConstantExpression(int value) {
    this.value = value;
  }
  public int evaluate() {
    return value;
  }
  public String toString() {
    return Integer.toString(value);
  }
}
```

Můžeme si vyzkoušet, že kód funguje dle očekávání.

```
Expression err = new Expression(); // nejde přeložit
Expression constant42 = new ConstantExpression(42);
constant42.evaluate(); // ==>  42
```

Všimněme si, že s hodnotou v proměnné `constant42` pracujeme jako s hodnotou typu `Expression`. To je v pořádku, protože konstanta je speciální případ celočíselného aritmetického výrazu. Když zavoláme metodu

[1] To je přirozené chování, protože neznáme kód všech metod.

`evaluate()`, použije se metoda definovaná ve třídě `ConstantExpression`, protože pracujeme s objektem této třídy.[2]

Zaměřme se nyní na jednotlivé aritmetické výrazy reprezentující aritmetické operace. Například operaci sčítání bychom mohli vyjádřit pomocí třídy:

```
public class AdditionExpression extends Expression {

  private final Expression rhs; // left hand side
  private final Expression lhs; // right hand side

  public AdditionExpression(Expression lhs, Expression rhs) {
    this.lhs = lhs;
    this.rhs = rhs;
  }
  public int evaluate() {
    return lhs.evaluate() + rhs.evaluate();
  }
  public String toString() {
    return "(" + lhs + " + " + rhs + ")";
  }
}
```

Pokud bychom měli jen operaci sčítání, takto pojatý kód by byl naprosto v pořádku. Avšak protože budeme potřebovat i třídy pro odčítání, násobení a dělení, vedle by to k velkému množství duplicitního kódu. Proto si zavedeme další abstraktní třídu `AbstractOperationExpression`, která bude společnou funkcionalitu zastřešovat. Tato třída by mohla mít následující kód:

```
public abstract class AbstractOperationExpression extends Expression {

  protected final Expression lhs;
  protected final Expression rhs;

  public AbstractOperationExpression(Expression lhs, Expression rhs) {
    super();
    this.lhs = lhs;
    this.rhs = rhs;
  }

  protected abstract String getOperationSymbol();
```

[2] Vytvořili jsme jej zavalonám `ConstantExpression(42)`.

```
  public String toString() {
    return "(" + lhs + " " + getOperationSymbol() + " " + rhs + ")";
  }
}
```

Tato třída má dva atributy `lhs` a `rhs` představující výrazy na levé a pravé straně operátoru a ty jsou nastaveny v konstruktoru. Všimněme si klíčového slova `protected` uvedeného u těchto atributů. To značí, že hodnota těchto atributů je přístupná z této třídy, případně jejich potomků.

Poznámka 1 *Pokud bychom místo* `protected` *použili* `public` *byly by atributy přístupné všem, což je nežádoucí vzhledem k tomu, že se chceme držet principu zapouzdření. Pokud bychom použili* `private` *byly by atributy přístupné pouze v třídě* `AbstractOperationExpression`*, což je opět nežádoucí, protože s těmito atributy budeme potřebovat pracovat v potomcích, kde jsou potřeba, abychom vrátili správnou hodnotu v metodě* `evaluate()`*.*

Dále ve třídě implementujeme metodu `toString()`, která se postará o převod výrazu do jeho textové reprezentace. Aby tato funkce fungovala bez úprav pro všechny potomky, zavedli jsme si abstraktní metodu `getOperationSymbol`, která vrací symbol dané operace a bude definována až v potomcích třídy `AbstractOperationExpression`. Tuto metodu jsme označili taktéž jako `protected`, aby k ní měla přístup jen aktuální třída a její potomci.[3]

Poznámka 2 *Počet abstraktních metod, které má* `AbstractOperationExpression` *stoupl na dvě. Protože tato třída zdědila ještě metodu* `evaluate` *od svého rodiče.*

Implementace konkrétních tříd je již přímočará a stačí nám definovat kód obou abstraktních metod.

```
public class AdditionExpression extends AbstractOperationExpression {

  public AdditionExpression(Expression lhs, Expression rhs) {
    super(lhs, rhs);
  }
  public int evaluate() {
    return lhs.evaluate() + rhs.evaluate();
  }
  protected String getOperationSymbol() {
    return "+";
  }
}
```

[3] Předpokládáme, že se jedná o metodu, kterou nikdo jiný nebude potřebovat.

Analogicky bychom postupovali pro zbývající operátory. V takto pojatých třídách definujeme opravdu jen to, co je vlastní jednotlivým operacím, a společný kód máme v abstraktní třídě, což usnadňuje budoucí úpravy či rozšíření.

2 Výčtové typy

Předchozí řešení představuje jen jeden možný pohled na řešení problému reprezentace celočíselných aritmetických výrazů. Dalo by se říct, že čtyři nebo pět tříd pro reprezentaci aritmetických výrazů je zbytečně mnoho.

Počet tříd bychom mohli zredukovat tak, že bychom nahradili abstraktní třídu `AbstractOperationExpression` a její potomky jednou třídou, která bude reprezentovat binarní operaci tak, že kromě atributů představujících levý a pravý operand bude obsahovat ještě identifikátor operace.

Tato třída by mohla tedy mít přibližně následující strukturu:

```
public class BinaryExpression extends Expression {

  private final Expression lhs;
  private final Expression rhs;
  private final ??? operation;
  // ...
}
```

Nabízí se otázka, jaký zvolit typ atributu `operation`. Mohli bychom použít řetězec (např. `"+"` pro sčítání, `"-"` pro odčítání, ...) nebo celé číslo (např. 0 pro sčítání, 1 pro odčítání, ...). Takové řešení však není příliš dobré, protože se nám v programu začnou objevovat *magické hodnoty*, hodnoty mající speciální význam. Takové hodnoty jsou problematické z několika důvodů. (i) Jednak komplikují jakékoliv další úpravy v kódu,[4] a (ii) může se stát, že ten, kdo bude náš kód používat, v dobré víře použije hodnotu, kterou jsme neočekávali a program nebude fungovat správně, např. prográmator bude předpokládat, že implementujeme operátor `"%"` pro zbytek po celočíselném dělení, ale náš kód s tímto operátorem nebude počítat, což povede chybnému výpočtu.

Právě pro takové situace obsahuje jazyk Java výčtové typy. Výčtový typ označuje konečnou předem definovanou množinu hodnot, např. dny v týdnu (pondělí, úterý, ...), měsíce (leden, únor, ...), světové strany (sever, jih, ...) nebo aritmetické operace (sčítání, odčítání, ...).

Výčtový typ se obvykle deklaruje podobně jako třída v samostatném souboru s tím, že použijeme klíčové slovo `enum` a použijeme výčet hodnot, jak ukazuje následující kód.

```
public enum ArithmeticOperation {
  ADDITION,
```

[4] Při změně je potřeba dohledat všechny výskyty dané hodnoty. Tento problém lze eliminovat důsledným používáním konstant.

```
  SUBTRACTION,
  MULTIPLICATION,
  DIVISION;
}
```

Poznámka 3 *Hodnoty výčtových typů se obvykle zapisují stejně jako konstanty, tj. velkými písmeny.*

Jednotlivé hodnoty výčtových typů jsou k dispozici jako konstantní statické atributy třídy, tj. `ArithmeticOperation.ADDITION` apod. Tyto hodnoty můžeme použít v příkazu/výrazu `switch-case`.

Díky tomu třída `BinaryExpression` může dostat konrétní podobu.

```
public class BinaryExpression extends Expression {

  private final Expression lhs;
  private final Expression rhs;
  private final ArithmeticOperation operation;

  public BinaryExpression(ArithmeticOperation operation, Expression lhs, Expression rhs) {
    super();
    this.lhs = lhs;
    this.rhs = rhs;
    this.operation = operation;
  }

  public int evaluate() {
    int lhsValue = lhs.evaluate();
    int rhsValue = rhs.evaluate();
    switch (operation) {
      case ADDITION: return lhsValue + rhsValue;
      case DIVISION: return lhsValue / rhsValue;
      case MULTIPLICATION: return lhsValue * rhsValue;
      case SUBTRACTION: return lhsValue - rhsValue;
      default:
        // jen kvuli prekladaci
        return 0;
    }
  }
}
```

Použití třídy je přímočaré:

```
Expression e = new BinaryExpression(ArithmeticOperation.ADDITION,
  new ConstantExpression(10), new ConstantExpression(20));
```

Pozorný čtenář si nejspíše povšiml, že třída `BinaryExpression` neumí vše, co třída `AbstractOperationExpression`, protože nedefinuje metodu `toString()`. V této metodě k určení symbolu operace můžeme použít konstrukci `switch-case` jako jsme učinili v metodě `evaluate()`, nebo můžeme informaci o symbolu operace svázat přímo s výčtovým typem.

Hodnoty výčtových typů jsou technicky taky objekty, které mohou mít atributy a metody, a které se deklarují ve stejném stylu jako v případě běžných tříd. Rozdíl je pouze v tom, že instance nevytváříme my, ale udělá to za nás překladač s běhovým prostředím. S použitím této vlastnosti bychom mohli upravit výčtový typ `ArithmeticOperation` následovně:

```
public enum ArithmeticOperation {
  ADDITION ("+"),
  SUBTRACTION ("-"),
  MULTIPLICATION ("*"),
  DIVISION ("/");

  private final String symbol;

  private ArithmeticOperation(String symbol) {
    this.symbol = symbol;
  }
  public String getSymbol() {
    return symbol;
  }
}
```

Všimněme si, že třída má jeden atribut `symbol`, který je soukromý a je to konstanta[5] a její hodnota je přístupná metodou `getSymbol()`. Dále má třída konstruktor, který se stará o nastavení atributu a musí být deklarovaný jako soukromý. Argumenty, které jsou předány konstrukturu se zapisují za jednotlivé hodnoty výčtu, viz řádek `ADDITION ("+")`, kde v závorkách jsou uvedeny argumenty konstruktoru.

S takto upraveným výčtovým typem můžeme snadno implementovat metodu `toString()`, kdy použijeme metodu `getSymbol()`, abychom získali odpovídající symbol operace.

```
public String toString() {
  return "(" + lhs + " " + operation.getSymbol() + " " + rhs + ")";
}
```

[5] Nechceme, aby vlastnosti hodnot výčtového typu šlu měnit.

3 Záznamy (třídy typu Record)

V praxi se velice často setkáváme se situací, kdy hlavní smyslem třídy je reprezentovat nějaké hodnoty, např. bod v prostoru, barvu (ve složkách RGB), informace o člověku, autu apod. Aby se práce s tímto druhem tříd usnadnila, byly do jazyka Java přidány třídy typu `Record` (záznamy). Deklarace těchto tříd se od ostatních v mnohém liší:

- atributy se uvádí do závorek za název třídy,
- konstruktor se vytvoří implicitně na základě seznamu atributů,
- hodnoty atributů jsou konstanty,
- a automaticky se vytvoří metody pro čtení hodnot atributů, ve tvaru `nazevAtributu()`,
- automaticky se vytvoří metody `equals`, `hashCode()` a `toString()`,
- třída může mít vlastní metody a může implementovat rozhraní,
- avšak nelze za třídy odvodit potomka.

Ukažme si několik příkladů:

```
/** osoba */
public record Person(String name, int age, double salary) { }

/** automobil */
public record Car(String regNo, String color) { }

/** bod v rovině */
public record Point(double x, double y) {
  public double distance(Point that) {
    double dx = this.x() - that.x();
    double dy = this.y() - that.y();
    return Math.sqrt(dx * dx + dy * dy);
  }
}
```

Použití těchto tříd je stejné jako u ostatních tříd, je však potřeba mít na paměti, že atributy nelze po vytvoření třídy měnit.

```
Person person01 = new Person("John Doe", 30, 45000);
// person01.name() // ==> "John Doe"
Car blueCar = new Car("1M12 358", "blue");
```

```
// blueCar.color() // ==> "blue"
Point p1 = new Point(10, 20);
Point p2 = new Point(20, 30);
p1.distance(p2); // ==> 14.142135623730951
```

S použitím záznamů, můžeme náš kód ještě přeformulovat. Celočíselné výrazy nebudeme definovat jako abstraktní třídu, ale zavedeme si rozhraní.[6]

```
public interface Expression {
  public int evaluate();
}
```

Třídy `ConstantExpression` a `BinaryExpression` můžeme převést na záznamy následovně.

```
public record ConstantExpression(int value) implements Expression {
  public int evaluate() {
    return value;
  }
}
```

```
public record BinaryExpression(ArithmeticOperation operation, Expression lhs,
    Expression rhs) implements Expression {

  public int evaluate() {
    int lhsValue = lhs.evaluate();
    int rhsValue = rhs.evaluate();
    switch (operation) {
      // ...
    }
  }
}
```

Záznamy si dobře rozumí s konstrukcí `switch-case`, která má pro ně širší možnosti. Ukážeme si to na implementaci metody `toString(Expression)`, která se liší od předchozí varianty v tom, že není definovaná jako běžná metoda, ale jako statická metoda, která pracuje s hodnotami atributů, které lze vyčíst pomocí veřejného rozhraní.

Tato metoda by mohla vypadat následovně:

```
  public static String toString(Expression expression) {
```

[6] Na rozhraní se dá nahlížet také jako na třídu, která má všechny metody abstraktní, avšak jazyk Java striktně rozlišuje abstraktiní třídy a rozhraní.

```
    switch (expression) {
    case ConstantExpression constExpr:
      return Integer.toString(constExpr.value());

    case BinaryExpression binaryExpression:
      return "(" + toString(binaryExpression.lhs()) + " "
        + binaryExpression.operation().getSymbol()
        + " " + toString(binaryExpression.rhs()) + ")";
    default:
        // kvuli prekladaci
        return "";
    }
  }
```

Všimněme si především, že za `case` je uveden typ a název proměnné. Pokud hodnota `expression` je zadaného typu, naváže se na zadanou proměnnou a začne se vykonávat příslušná větev výpočtu. Pokud by hodnota `expression` byla typu `ConstantExpression` naváže se na proměnnou `constExpr`, se kterou se v první větvi výpočtu pracuje. Analogicky to platí i pro hodnoty typu `BinaryExpression`.

Pokud je v `case` použit typ `Record`, můžeme si nechat záznam rozbít na jednotlivé složky následujícím způsobem:

```
switch (expression) {
    case ConstantExpression(int value):
      return Integer.toString(value);
    case BinaryExpression(ArithmeticOperation operation, Expression lhs, Expression rhs):
      return "(" + toString(lhs)  + " " + operation.getSymbol() + " " + toString(rhs) + ")";
    default: return ""
}
```

Konstrukce `switch-case` má ještě jednu vlastnost, která umožňuje otestovat, zda předaný objekt splňuje zadanou podmínku. Toho docílíme pomocí klíčového slova `when` zapsaného za daný typ. Následující příklad ukazuje, jak lze zvýraznit dělení nulou:

```
switch (expression) {
//...
  case BinaryExpression(ArithmeticOperation operation, Expression lhs, Expression rhs)
      when ((operation == ArithmeticOperation.DIVISION)
            && (rhs instanceof ConstantExpression constExpr)
            && (constExpr.value() == 0)):
        return "DELENI NULOU";
```

```
    case BinaryExpression(ArithmeticOperation operation, Expression lhs, Expression rhs):
      return "(" + toString(lhs)  + " " + operation.getSymbol() + " " + toString(rhs) + ")";
}
```

Pokud je `expression` typu `BinaryExpression`, operace je dělení a současně druhý operand je konstanta s hodnotou 0, vrátí se text `"DELENI NULOU"`, v opačném případě se pokračuje další větví výpočtu.

4 Omezení dědičnosti

Jsou situace, kdy chceme mít pod kontrolou, kdo a jak může s pomocí dědičnosti vytvářet odvozené třídy z našich tříd. Například můžeme chtít, aby z dané třídy nešel vytvořit potomek, protože chceme garantovat nějakou vlastnost (například neměnnost) nebo nechceme, aby vznikla třída, která bude mít jiné než námi definované chování (například ve více vláknových aplikacích). V takovém případě můžeme v deklaraci třídy uvést klíčové slovo `final`, které zajistí, že ze třídy nepůjde odvodit potomek, např.

```
public final class AdditionExpression extends AbstractOperationExpression { /* ... */ }
```

Při omezování dědičnosti můžeme být ještě striktnější a určit výčet tříd, které mohou z námi definované třídy dědit nebo implementovat zadané rozhraní.

Vezměme si jako příklad metodu `toString(Expression)`, kdy předpokládáme, že jako argument je předána hodnota typu `ConstantExpression` nebo `BinaryExpression`. Nikdo nám však negarantuje, že nevznikne jiná třída implementující rozhraní `Expression` a nepředá ji naší metodě.

Aby se takovým situacím předešlo, má jazyk Java možnost vytvořit *zapečětěné třídy* (*sealed classes*), které definují, které třídy z nich mohou dědit. Zapečetěné třídy musí být označeny klíčovým slovem `sealed` a za klíčovým slovem `permits` obsahují seznam tříd, které dědí z této třídy.[7]

V našem příkladu by to vedlo k deklaraci:

```
public sealed interface Expression permits ConstantExpression, BinaryExpression { ... }
```

U takto upraveného kódu, můžeme v metodě `toString(Expression)` bezpečně vypustit větev `default`, protože je garantované, že rozhraní `Expression` implementují jenom naše dvě třídy, a nemůže proto existovat další větev výpočtu.

[7] Ty musí být označený jako `final` nebo `sealed`.