Dědičnost

5

Při návrhu programu v objektově orientovaném programovacím jazyce je důležité správně identifikovat jednotlivé třídy, jejichž objekty budou v programu vystupovat, a určit jejich atributy a metody. Řekněme, že bychom chtěli vytvořit informační systém pro školu. Určitě by v takovém programu měly být (mimo jiné) objekty tříd *žák*, *učitel* a *technický pracovník*. Zamyslíme-li se nad tím, jaké atributy by tyto objekty měly mít, zjistíme, že některé atributy (např. *jméno*, *datum narození*, *bydliště*) se budou vyskytovat u všech těchto objektů, bez ohledu na to, do které třídy objekt patří. Jiné atributy budou specifické pro objekty třídy *žák* (*známky z předmětů*, *zameškané hodiny*, apod.), další atributy budou specifické pro učitele (*aprobace*, *mzda*, *platová třída*), podobně specifické atributy budou mít objekty třídy *technické pracovník* (*profese*, *mzda*, *platová třída*). Kdybychom pokračovali v analýze do větší hloubky a zajímali se o to, jaké metody by naše objekty měly mít, určitě bychom dospěli k tomu, že některé metody se budou v jednotlivých třídách opakovat. Například výplata mzdy u učitele bude probíhat stejně jako u technického pracovníka.

Opakování stejného kódu patří k velkým programátorským prohřeškům.[1] Abychom mu v této situaci předešli, bylo by rozumné zavést třídu *zaměstnanec*, která by pokrývala všechnu společnou funkcionalitu týkající se zaměstnanců, tj. tříd *učitel* a *technický pracovník*.

Poznámka 1 *Toto je přirozené rozhodnutí, protože v této situaci je pojem zaměstnanec obecnější než pojem učitel, podobně je zaměstanec obecnější pojem než technický pracovník. Důležité je, že cokoliv platí pro zaměstnance, bude platit i pro učitele a technického pracovníka. Opačně to však neplatí. Učitel je speciální případ zaměstnance a může mít další specifické atributy a metody, podobně jako technický pracovník.*

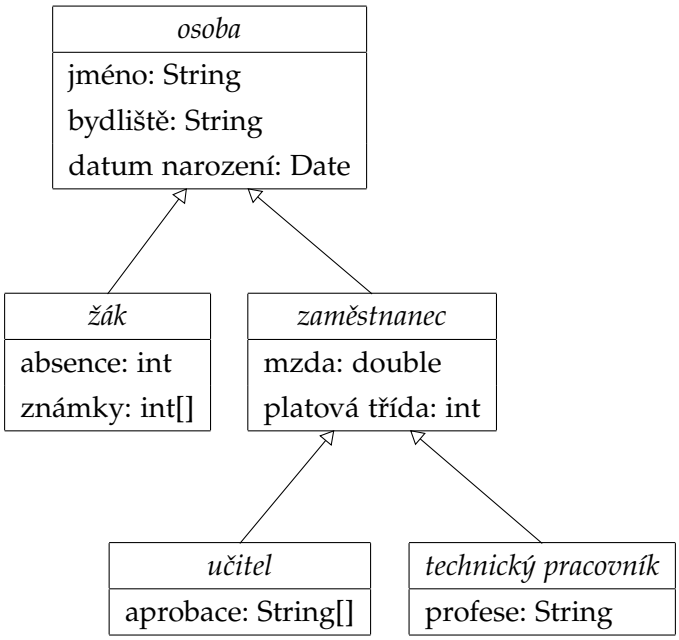
Máme tedy obecnou třídu *zaměstnanec*, ale ještě potřebujeme aparát, který nám umožní sdílet funkcionalitu mezi třídami *zaměstnanec* a *učitel* (resp. *technický pracovník*). Tímto aparátem je v jazyce Java *dědičnost*.

Dědičnost nám umožňuje určit, že třída (např. *učitel*) je specifickým případem obecnější třídy (*zaměstnanec*). V takovém případě tato specifická třída (*učitel*) přebírá veškerou funkcionalitu z třídy obecnější (*zaměstnanec*), přičemž může tuto odvozenou funkcionalitu dále rozšířit nebo pozměnit. Jelikož je tento mechanismus hodně podobný předávání genetických informací v biologickém slova smyslu, bývá obecnější třída označována jako *rodič* (nebo *rodičovská třída*) a třída z ní odvozená jako *potomek*.

Dědičnost se nemusí omezovat jen na jednoduchý vztah rodič-potomek. Jednotlivé třídy z pohledu dědičnosti často tvoří celé hierarchie. Vraťme se proto k našemu úvodnímu příkladu. Jelikož některé vlastnosti jsou

[1] Je to práce navíc, v opakujícím se kódu se špatně orientuje a hlavně, chceme-li provést úpravu takového programu, nestačí to udělat jedenkrát na jednom místě, ale je potřeba opravit všechna místa, kde se kód opakuje. To je opět práce navíc, ale také potenciální zdroj chyb.

společné žákům i zaměstnancům, nabízí se zavést obecnou třídu *Osoba*, která tyto vlastnosti bude popisovat a zastřešovat. Nástin takové hierarchie ukazuje Obrázek 1.



Obrázek 1: Možný vztah dědičnosti mezi třídami z úvodního příkladu; šipky zároveň ukazují vztah mezi konkrétními a obecnějšími třídami objektů.

1 Dědičnost v Javě

Dědičnost z pohledu programovacího jazyka Java znamená, že potomek přebírá od svého rodiče minimálně všechny veřejné (`public`) atributy a metody, tj. má přinejmenším stejné metody jako rodič, přičemž může chování těchto metod změnit, je-li to potřeba. Potomek pochopitelně může mít další, své vlastní, atributy a metody. My už jsme se s dědičností setkali. Každá třída v Javě je totiž buď přímým nebo nepřímým potomkem třídy `Object`, která nabízí nejzákladnější metody pro práci s objektem. Jednou z takových metod je metoda `toString()` vracející textovou informaci o daném objektu.

Vytvořme si třídu `Car`, nemusí obsahovat žádnou metodu či atribut.

```
public class Car { }
```

Neuvedeme-li žádnou informaci o rodičovské třídě, automaticky se uvažuje, že třída `Car` je potomkem třídy `Object`. Můžeme proto bez problémů vytvořit objekt této třídy, a pak zavolat jeho metodu `toString()`.

```
Car car = new Car();
System.out.println(car.toString());
// vypise priblizne: Car@6d06d69c
```

Metoda z rodičovské třídy `Object` se nám postará o vypsání (implicitní) informace o objektu. Kdybychom prezentovanou informaci objektu chtěli změnit, stačí nám definovat vlastní metodu `toString()` a ta chování původní metody překryje.

```
public class Car {
  public String toString() {
    return "jsem auto";
  }
}
```

Pokud objekt změněné třídy použijeme, měli bychom vidět změněné chování.

```
Car car = new Car();
System.out.println(car.toString());
// vypíše "jsem auto"
```

Ukažme si důmyslnější příklad dědičnosti, opět s automobily. Pro naši potřebu si vytvoříme třídu `Vehicle`, která bude popisovat pohyb (obecného) dopravního prostředku. Objekty této třídy budou mít dva atributy `x`, `y` udávající souřadnice, kde se dopravní prostředek nachází,[2] a atribut `direction` určující směr,[3] kterým se bude pohybovat a dvě metody: (1) `turn(int angle)`, která natočí dopravní prostředek o daný úhel doleva nebo doprava podle použitého znaménka a (2) metodu `driveForward(double dist)`, která posune dopravní prostředek o daný počet kilometrů v nastaveném směru.

```
public class Vehicle {
  private int direction;
  private double x;
  private double y;

  public Vehicle(double x, double y) {
    this.x = x;
    this.y = y;
    this.direction = 0;
  }

  public void turn(int angle) {
    this.direction += angle;
  }
```

[2] Pro jednoduchost předpokládáme, že Země je placatá a popsatelná dvěma kartézskými souřadnicemi.

[3] Používáme úhel s osou x.

```
  public void driveForward(double distance) {
    double angle = Math.toRadians(direction);
    x += Math.cos(angle) * distance;
    y += Math.sin(angle) * distance;
  }

  public double getX() { return x; }
  public double getY() { return y; }

  public String toString() {
    return "Vehicle at " + x + ", " + y;
  }
}
```

Pro nás nebude zajímavá ani tak třída `Vehicle`, ale třídy, které si z ní odvodíme. Vytvoříme si dvě třídy, třídu `Car` představující osobní automobil, která bude mít atribut udávající počet pasažérů aktuálně cestujících v automobilu, a třídu `Truck` představující nákladní automobil s atributy popisujícími náklad.

Tyto třídy jsou nastíněny v následujících kódech.

```
1  public class Car extends Vehicle {
2    private int passengers;
3
4    public Car(double x, double y, int p) {
5      super(x, y);
6      this.passengers = p;
7    }
8
9    public String toString() {
10     return "Car at " + getX() + ", " + getY() + " with " + passengers + " passengers";
11   }
12 }
```

```
1  public class Truck extends Vehicle {
2    private String payload;
3    private int amountOfPayload;
4
5    public Truck(double x, double y, String payload, int amountOfPayload) {
6      super(x, y);
7      this.payload = payload;
8      this.amountOfPayload = amountOfPayload;
```

```
  }

  public String toString() {
    return "Truck at " + getX() + ", " + getY() + " with " + amountOfPayload + " items of
    ↪  " + payload;
  }
}
```

Obě deklarace jsou podobné deklaracím tříd, se kterými jsme se dosud setkávali. Rozdíl je v tom, že za názvem třídy uvádíme klíčové slovo `extends` a název rodičovské třídy. Dále v konstruktoru pomocí klíčového slova `super` uvádíme, jak by se vytvořil objekt rodiče z předaných argumentů. V našem případě předáváme počáteční souřadnice. Volání `super(...)` musí být první věc, kterou v konstruktoru provedeme. Pokud volání `super` neuvedeme, překladač zavolá konstruktor `super()`, rodič však takový konstruktor musí mít, jinak překladač oznámí chybu.

S objekty tříd `Car` i `Truck` můžeme pracovat běžným způsobem, vytvářet je a volat jejich metody.

```
Car car = new Car(0, 0, 3);
car.driveForward(2);
car.turn(45);
car.driveForward(1);
System.out.println(car);
// Car at 2.707.., 0.707... with 3 passengers
```

Při práci s objekty, jež jsou instancemi třídy, která je potomkem nějaké obecné třídy, můžeme využít toho, že potomek přebírá veškeré metody od svého rodiče a má tedy stejné rozhraní (v širším slova smyslu). To nám umožňuje pracovat s objekty různých tříd, které mají společného rodiče, uniformním způsobem. Ukažme si to na následující metodě.

```
static void driveAround(Vehicle vehicle) {
  for (int i = 0; i < 4; i++) {
    vehicle.turn(90);
    vehicle.driveForward(1);
    System.out.println(vehicle);
  }
}
```

Tato metoda bere jako svůj argument objekt třídy `Vehicle`, ale díky tomu, že objekty třídy `Car` a `Truck` mají stejné rozhraní, můžeme tyto objekty předat také.

```
driveAround(new Car(0, 0, 2));
driveAround(new Truck(0, 0, "books", 100));
```

Všimněte si, že toto chování je podobné polymorfismu, jak jsme si jej představili v minulém semináři, avšak nyní nevyužíváme rozhraní (v úzkém slova smyslu), ale vztah dědičnosti mezi třídami a sdílení rozhraní v širším slova smyslu.

V předchozích případech jsme využívali toho, že potomek přejímá funkcionalitu svého rodiče. Avšak v případě potřeby může potomek toto chování pozměnit, aby odpovídalo jeho potřebám. Ukážeme si to na třídě `Taxi`, která bude představovat osobní automobil, u něhož podle počtu ujetých kilometrů bude evidován výdělek.

```
1  public class Taxi extends Car {
2    private static final int FARE = 28;
3    private double revenue = 0.0;
4
5    public Taxi(double x, double y, int passengers) {
6      super(x, y, passengers);
7    }
8
9    public void driveForward(double distance) {
10     revenue += distance * FARE;
11     super.driveForward(distance);
12   }
13
14   public double getRevenue() {
15     return revenue;
16   }
17 }
```

Třída je potomkem třídy `Car` a má navíc atributy `FARE` (taxa za kilometr) a `revenue` (tržba). Zajímavá je metoda `driveForwad`. Tato metoda na řádku 10 spočítá jízdné a přičte jej k tržbě. Volání `super.driveForward` na řádku 11 zavolá metodu z rodiče a zajistí, že metoda `driveForward` se bude chovat stejně, jako bychom ji převzali od rodiče. Klíčové slovo `super` nám říká, že chceme volat metodu předka. Pokud bychom jej neuvedli a volali bychom pouze `driveForward`, dostali bychom se do nekonečné smyčky, protože bychom volali tu samou metodu. Můžete si vyzkoušet jako cvičení.

Poznámka 2 *Díky tomu, že třída* `Taxi` *je potomkem třídy* `Car` *a nepřímo i potomkem třídy* `Vehicle`*, můžeme objekty třídy* `Taxi` *použít i tam, kde se očekává objekt třídy* `Vehicle`*, protože má stejné rozhraní. Vyzkoušejte si to s metodu* `driveAround`*.*

Poznámka 3 *Jazyk Java umožňuje pouze tzv. jednoduchou dědičnost, což znamená, že každá třída má právě jednoho rodiče. Neuvedeme-li jej, automaticky se uvažuje jako předek třída* `Object`*.*

1.1 Skládání objektů vs. dědičnost

Již umíme navrhovat třídy objektů dvěma způsoby. (1) V předchozích seminářích jsme si ukázali skládání objektů, (2) nyní známe i dědičnost. Který z těchto způsobů zvolit při vytváření nové třídy hodně závisí na konkrétní situaci a vlastnostech tříd. Pokud je nová třída speciálním případem jiné, nabízí se použít dědičnost, v opačném případě je vhodnější použít skládání objektů.

Při rozhodování nám mohou pomoci dvě mnemotechnické pomůcky: *is-a* a *has-a*. Pokud nám dává smysl říct (v angličtině) `Car` *is a* `Vehicle`, dává smysl použít dědičnost. Na druhou stranu, pokud lze říct, že `Car` *has an* `Engine`, měli bychom použít skládání.

Tyto pomůcky ale nejsou všespásné a záleží na konkrétní situaci. Vezměme si například třídu `Rectangle`, kterou jste měli za úkol vytvořit v jednom z předešlých seminářů.

```
public class Rectangle {
  private int a;
  private int b;

  public Rectangle(int a, int b) { /* ... */ }
  public int getArea() { /* ... */
  public void printOut() {/* ... */ }
}
```

Kdybychom chtěli vytvořit třídu `Square` reprezentující čtverec, dávalo by smysl odvodit ji z třídy `Rectangle`, protože si můžeme říct `Square` *is a* `Rectangle`. Vyzkoušejte si vytvořit třídu:

```
public class Square extends Rectangle {
  public Square(int a) {
    super(a, a);
  }
}
```

Co kdyby ale třída `Rectangle` obsahovala i metody:

```
public void setA(int a) { this.a = a; }
public void setB(int b) { this.b = b; }
```

V takovém případě by šlo udělat:

```
Square s = new Square(10);
s.setA(3);
s.setB(5);
s.printOut(); // vykreslí se obdélník
```

Toto není výsledek, který bychom očekávali, protože čtverec by měl vždy býti čtvercem. Problém tkví v tom, že v tomto případě[4] třída `Square` není speciálním případem `Rectangle`. Metody třídy `Rectangle` umožňují nastavit velikost obou stran, což je vlastnost, kterou by třída `Square` mít neměla. Proto při rozhodování o použití dědičnosti výše zmíněná pomůcka nemusí stačit, a je potřeba se dobře zamyslet nad tím, jestli chování jednotlivých metod je v souladu s dědičností.

2 Použití dědičnosti a polymorfismu

2.1 Porovnání objektů

Dříve jsme si ukázali, že operátorem `==` můžeme porovnat, zda dva výrazy (nejčastěji proměnné) odkazují na stejný objekt. Proto, máme-li dva odlišné objekty se shodnými hodnotami atributů, bude výsledek jejich porovnání pomocí `==` vždy roven `false`.

```
Car car1 = new Car("0M0 6502", "green");
Car car2 = new Car("0M0 6502", "green");

(car1 == car2)          // false
```

Toto ale často není chování, které bychom potřebovali, a často narazíme na problém, jak zjistit, zda se dva objekty shodují ve svých atributech a můžeme o nich prohlásit, že si jsou rovny. S tím nám pomůže dědičnost.

Právě pro tyto účely má třída `Object` metodu `equals(Object)`, která se volá, pokud potřebujeme ověřit, zda se náš objekt shoduje v atributech s jiným objektem. Připomeňme, že každá třída v jazyce Java je buď přímým, či nepřímým potomkem třídy `Object`, proto se můžeme spolehnout, že každý objekt má tuto metodu. Výchozí chování metody `equals` je shodné s porovnáním pomocí operátoru `==`, avšak díky dědičnosti můžeme toto chování předefinovat.

Vezměme si například třídu `Car` se třemi atributy `plateNo`, `color` (oba typu `String`) a `speed` (typu `int`), kterou průběžně v našem semináři používáme. Metoda `equals` by v hrubých obrysech mohla vypadat následovně.

```
public boolean equals(Object other) {
  return (this.plateNo == other.plateNo)
     && (this.color == other.color)
     && (this.speed == other.speed);
}
```

Toto řešení je hned z několika důvodů špatné a nebude fungovat. První problém tkví v tom, že o objektu, který byl předán argumentem `other` víme jen to, že je potomkem třídy `Object`. Nejsme schopni říct jaké má

[4] Po přidání metod `setA` a `setB`.

všechny atributy nebo metody. Potřebovali bychom mít nějaký nástroj, který nám ověří, že objekt je instancí třídy `Car`. Pro tyto účely má jazyk Java operátor `instanceof`, který vrací `true`, pokud je objekt instancí dané třídy nebo implementuje zadané rozhraní, jinak vrací `false`.

```
Car car = new Car("0M0 6502", "green");
Light light = new Light("blue");
(car instanceof Car)           // true
(car instanceof Light)         // false
(light instanceof Switchable)  // true
```

Můžeme proto použít operátor `instanceof` k ověření, že v argumentu `other` máme objekt třídy `Car`.

```
public boolean equals(Object other) {
  if (other instanceof Car) {
    return (this.plateNo == other.plateNo)
    // ...
  } else return false;
}
```

Toto ověření se provádí až za běhu programu. Proto, abychom mohli metodu přeložit, potřebujeme ještě překladač instruovat, aby s objektem, který je v proměnné `other`, pracoval jako s objektem třídy `Car`. K tomu slouží *explicitní přetypování*. Tato operace se v jazyce Java zapisuje jako `(Typ)obj` a říká, že s objektem `obj` má být nakládáno jako s objektem třídy `Typ`.[5] Dodejme, že to je možné pouze, pokud `obj` je opravdu objektem třídy `Typ` nebo některého z jejích potomků. Pokud explicitní přetypování nelze provést, je vyvolána výjimka, a proto se přetypování většinou používá ve spojení s operátorem `instanceof`.

Mohlo by se zdát, že nám již nic nebrání v tom, abychom vzali kód z úvodního příkladu a naprogramovali plně funkční metodu pro porovnání dvou objektů třídy `Car`. Úvodní kód ale obsahuje ještě jednu záludnou chybu, na kterou nás překladač neupozorní, a projeví se jen za určitých okolností. Připomeňme, že jednotlivé řetězce jsou také objekty, a chceme-li porovnat řetězce, zdali obsahují stejný text, musíme použít metodu `equals` místo operátoru `==`.

Výsledná metoda porovnávající, jestli jsou si dva objekty třídy `Car` rovny, by mohla vypadat následovně.

```
public boolean equals(Object other) {
  if (other instanceof Car) {
    Car otherCar = (Car) other;
    return (this.plateNo.equals(otherCar.plateNo))
         && (this.color.equals(otherCar.color))
         && (this.speed == otherCar.speed);
  } else return false;
}
```

[5] Případně s objektem implementujícím rozhraní `Typ`.

Protože se s podobnými konstrukcemi dá v programech setkat velice často, existuje i stručnější varianta, kdy název proměnné uvedeme jako třetí operand operátoru `instanceof` a vypustíme explicitní přetypování. Použití stručnější varianty vypadá následovně:

```
public boolean equals(Object other) {
  if (other instanceof Car otherCar) {
    return (this.plateNo.equals(otherCar.plateNo))
         && (this.color.equals(otherCar.color))
         && (this.speed == otherCar.speed);
  } else return false;
}
```

Poznámka 4 *Ukázkový kód pro porovnání není dokonalý a v některých případech nebude fungovat dle očekávání, např. pokud porovnáme instanci třídy s jejím potomkem. Je proto dobré si metodu* `equals` *nechat vygenerovat v IDE.*

2.2 Dědičnost vs. rozhraní

Všimněme si, že dědičnost a rozhraní se v některých ohledech nápadně podobají a v jiných se zcela odlišují. Rozhraní slouží primárně k popsání toho, jaké metody daná třída má, a tedy nám garantuje, že je můžeme použít, díky čemuž můžeme pracovat s různými třídami jednotným způsobem. Všimněme si taktéž, že třída může implementovat několik různých rozhraní.

Dědičnost nám vedle toho zajišťuje, že potomek může být použit na místě, kde se očekává jeho předek, protože potomek má minimálně stejné metody jako jeho předci. V tomto směru je dědičnost méně univerzální než rozhraní, protože polymorfismus funguje pouze mezi příbuznými objekty, což je dále omezeno skutečností, že každá třída má pravě jednoho předka. Avšak je nutné dodat, že dědičnost nám umožňuje sdílet funkcionalitu (kód, případně data) směrem od rodiče k potomkovi a tím nám usnadnit programování a omezit duplicitní kód.

Toto je výchozí úvaha, které bychom se měli držet při rozhodování, jestli použít dědičnost nebo rozhrání. Vedle toho existuji mechanismy, který rozdíly mezi dědičností a rozhraními stírají. Jednak jsou to abstraktní třídy, kde některé metody nejsou implementovány (podobně jako u rozhraní), těm se budeme věnovat v dalším semináři. A dále jsou to výchozí metody (*default methods*), které umožňují v rozhraní přímo určit, jak se má metoda chovat, pokud ji implementující třída nepřepíše.

V minulém semináři jsme pracovali s rozhraním `Shape`:

```
/** Rozhrani objektu rovinnych geom. utvaru */
public interface Shape {

  /** vypise informace o geom. utvaru */
  public void printOut();
```

```
  /** vrati plochu geom. utvaru */
  public double getArea();
}
```

Mohli bychom chtít, aby metoda `printOut`, pokud není přepsána implementující třídou vypsala, alespoň nějakou informaci. V takovém případě metodu označím modifikátorem `default` a uvedeme její tělo.

```
default public void printOut() {
  System.out.println("Jsem tvar.");
}
```

Případně můžeme volat metody daného rozhraní:

```
default public void printOut() {
  System.out.println("Jsem tvar s plochou:" + getArea());
}
```

Úkoly k procvičení 1 *Vyzkoušejte si použití výchozích metod tak, že z implementujících tříd odstraníte (zakomentujete) metody* `printOut`.

Poznámka 5 *Výchozí implementace metod koncepčně nezapadají do objektového modelu jazyka Java, ale byly do něj přidány z praktických důvodů. Umožňují totiž přidat do rozhraní nové metody tak, aby se nerozbil stávající kód. Pokud bychom měli rozhraní a přídali do něj novou metodu, znamenalo by to, že by všichni, kdo toto rozhraní používají, museli dopsat implementaci pro novou metodu, což nemusí být reálně proveditelné. Například ve verzi 8 jazyka Java došlo k celé řadě zásadních změn, které rozšiřovaly existující rozhraní, což by vedlo k tomu, že všechen do té doby napsaný kód by přestal fungovat. Tento problém právě vyřešily výchozí implementace metod, která zajistili, že starý kód může fungovat s rozšířenými rozhraními.*