Objektově orientované programování 2

Ústředním pojmem objektově orientovaného programovaní, jak již název napovídá, je *objekt*. Objekty v paměti počítače modelují jednotlivé objekty z reálného světa. Stejně jako každý objekt v reálném světě má vlastnosti, které jej charakterizují (např. barvu, tvar nebo jméno), mají i objekty v paměti počítače uložené své vlastnosti, které je charakterizují. Těmto vlastnostem budeme říkat *atributy*.

Dva automobily, u nichž by nás zajímala jejich registrační značka, barva a aktuální rychlost, můžeme mít v paměti počítače uložené následovně.

reg. značka:	1A2 4816
barva:	modrá
rychlost:	55

reg. značka:	0M1 1235
barva:	červená
rychlost:	120

Pokud bychom chtěli vytvořit program na evidenci žáků ve škole, množina atributů by zcela jistě byla jiná, například *jméno, datum narození, třída, známky z jednotlivých předmětů*, atd. Každý objekt by představoval jednoho žáka a naše objekty by mohly vypadat následovně.

jméno:	Jan Majer
třída:	3.B
dat. nar.	1.2.2007
čj:	1
ma:	2
aj:	3

jméno:	Marie Hermanová
třída:	3.C
dat. nar.	15.6.2006
čj:	2
ma:	1
aj:	2

Jaké budou mít jednotlivé objekty atributy, je čistě na nás a našich potřebách. Všimněte si však, že v našich příkladech jsme měli dva typy objektů, *automobil* a *žák*, přičemž všechny objekty stejného typu mají stejné atributy, i když hodnoty těchto atributů se liší.

Skutečnost, že objekty mají stejné vlastnosti, popisujeme pomocí tzv. *tříd*. Třídu můžeme chápat jako předpis toho, jak vytvořit nějaký objekt a popis toho, jaké má vlastnosti. Říkáme proto, že *objekt je instancí třídy*.

V jazyce Java je každý objekt vždy instancí nějaké třídy. Vrátíme-li se k našemu příkladu, můžeme uvažovat třídu *automobil*, která říká, že automobil má vlastnosti (i) registrační značka, (ii) barva a (iii) aktuální rychlost. Modrý automobil s registrační značkou 1A2 4816, modré barvy jedoucí rychlostí 55 km/h je jednou instancí třídy *automobil*, červený automobil s registrační značkou 0M1 1235 jedoucí rychlostí 120 km/h je další instancí této třídy.

Poznámka 1 *Vztah mezi třídou a objektem můžeme také chápat podobně jako vztah mezi výkresem a finálním výrobkem, vyrobeným podle tohoto výkresu. Výkres (třída) nepředstavuje finální výrobek, ale přesně popisuje, jak objekt bude vypadat a jak jej vytvořit. Na základě tohoto popisu pak můžeme vytvořit libovolné množství výrobků (objektů).*

Zatím jsme uvažovali pouze objekty a jejich vlastnosti, tedy data, se kterými má program pracovat. Aby náš popis byl úplný, potřebujeme ještě nějaký prostředek, který s těmito daty může pracovat. Tím jsou v objektově orientovaných jazycích *metody*. Metodu můžeme chápat jako posloupnost příkazů, která pracuje s atributy objektu, tj. čte nebo mění jejich hodnoty. Metody jsou deklarovány ve třídě a jejich smyslem je popsat chování objektů dané třídy. Můžeme se na to dívat také jako na povel, co má daný objekt provést. V případě objektů třídy *automobil* by dávaly smysl například metody *zrychli* nebo *brzdi*. Jednotlivé metody mohou mít parametry, které blíže specifikují operaci s objektem, např. o kolik km/h se má zrychlit. Metody mohou mít návratovou hodnotu, pomocí níž je možné získat informace o objektu nebo o výsledku operace.

Důležité je, že s objektem, tj. s hodnotami jeho atributů se manipuluje zásadně pomocí metod. Má to několik důvodů. (i) Metoda nám umožňuje jednoduše vyjádřit, co se s objektem má dít (např. zrychli o 10 km/h.) (ii) Metody nám zajistí, že objekt se nemůže dostat do nekorektního nebo nekonzistentního stavu (např. automobil nepojede více než maximální rychlostí, rychlost nebude záporná). (iii) Přístup přes metody usnadní případné budoucí úpravy v kódu. Například, pokud bychom jméno studenta nechtěli mít uložené jako jeden řetězec, ale chtěli jej rozdělit do dvou atributů – jméno a příjmení, stačí nám upravit jen metody, které přistupují přímo k těmto atributům. Obvykle je takový zásah rychlý, protože je potřebný jen na pár snadno identifikovatelných místech.

Skrývání atributů před přístupem z vnějšku objektu je jeden ze základních principů objektového programování a označuje se jako *zapouzdření*.

Poznámka 2 *Díky dodržování principu zapouzdření můžeme snadno změnit to, jak je objekt uvnitř implementován (např. můžeme použít jinou efektivnější reprezentaci dat nebo použít efektivnější algoritmus), aniž bychom museli jakkoliv zasahovat do ostatního kódu.*

Každá metoda by měla dělat vždy jen jednu jasně definovanou věc, aby byl program srozumitelnější. Proto obvykle máme metody, které buď čistě mění vlastnosti objektu, nebo čistě vrací nějakou informaci o daném objektu. Princip zapouzdření, strukturu objektu a komunikaci mezi objekty nastiňuje Obrázek 1.

Aby objekt mohl být vždy v konzistentním stavu, musí mít hodnoty atributů v konzistentním stavu již od svého vytvoření. O počáteční nastavení atributů se stará tzv. *konstruktor*, metoda, jež je zavolána při vytvoření nového objektu v paměti počítače. Konstruktor může mít parametry, kterými lze říct, jak se mají atributy na počátku nastavit.

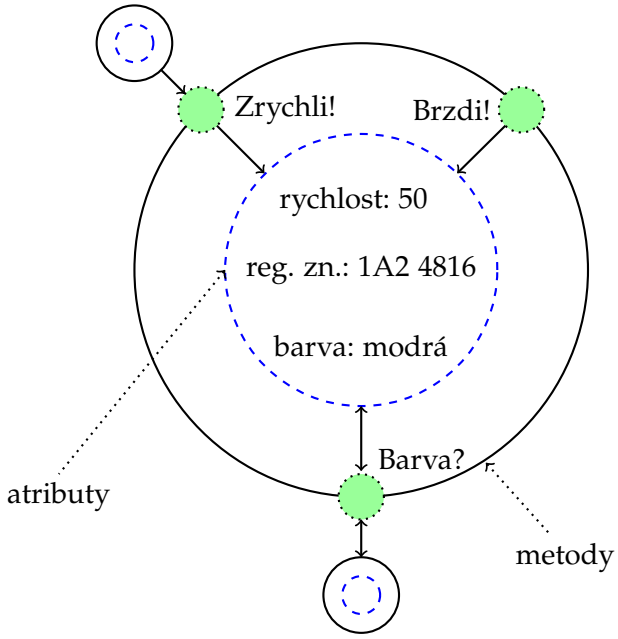
1 Objektově orientované programování v Javě

Všechny tyto pojmy si nyní ukážeme na třídě *automobil* z našeho příkladu.[1]

```
1  public class Car {
2
3    // deklarace atributů
4    private final String plateNo;
5    private final String color;
6    private int speed;
7
8    // konstruktor
9    public Car(String plateNo, String color) {
10     this.plateNo = plateNo;
11     this.color = color;
12     this.speed = 0;
13   }
14
15   // metody
16   public void accelerate(int s) {
17     speed += s;
18   }
19
20   public void brake(int s) {
21     speed -= s;
22     if (speed < 0) speed = 0;
23   }
24
25   public int getSpeed() {
26     return speed;
27   }
28
29   public void printOut() {
30     System.out.println(color + " car no. " + plateNo + " is running " + speed + "km/h");
31   }
32 }
```

Na řádku 1 deklarujeme třídu `Car`. Tato třída, resp. její zdrojový kód, musí být uložena v souboru `Car.java`, jak jsme si uvedli dříve. Řádky 4 až 6 představují deklarace atributů dané třídy, ty jsou podobné deklaraci

[1] Protože jazyk Java má zažité konvence pro pojmenovávání proměnných, metod, tříd, které vychází z anglického jazyka a některé nástroje s těmito konvencemi počítají, budeme i my v kódu používat anglické názvy.



Obrázek 1: Ilustrace zapouzdření a komunikace mezi objekty

proměnných. Atributy `plateNo` a `color` jsou typu String (řetězec) a jsou označany jako `final`, což znamená, že jejich hodnoty jsou nastaveny při inicializaci objektu a už je nepůjde změnit. Před každým atributem je uvedeno klíčové slovo `private`, které značí, že daný atribut je přístupný pouze metodám dané třídy.[2]

Konstruktor objektu je popsán na řádcích 9 až 13. Klíčové slovo `public` udává, že objekt půjde vytvořit z jakéhokoliv jiného objektu. Následuje název třídy a seznam parametrů konstruktoru. Tento seznam se skládá z dvojic `<typ> <parametr>` oddělených čárkou. Tělo konstruktoru se, stejně jako u běžných metod, zapisuje do složených závorek `{ }`.

V těle konstruktoru (řádky 10 až 12) nastavíme hodnoty jednotlivých atributů. Běžně se k jednotlivým atributům přistupuje čistě pomocí jejich názvu, jako by to byla proměnná. V našem konstruktoru se ale názvy obou parametrů shodují s názvy atributů, které chceme nastavovat. Proto jsme použili klíčové slovo `this`, které odkazuje na aktuální objekt. Řádkem `this.plateNo = plateNo` jsme řekli: *„atributu* `plateNo` *tohoto objektu přiřaď hodnotu parametru* `plateNo`*."* V případě atributu `speed` bychom `this.` mohli vypustit, protože zde nedochází ke kolizi jména atributu a parametru metody.

Řádky 16 až 31 definují jednotlivé metody třídy `Car`. První metoda `accelerate` má zajistit zvýšení rychlosti automobilu. Klíčové slovo `public` udává, že danou metodou mohou použít (zavolat) další objekty.[3] Následuje typ návratové hodnoty. Typ `void` udává, že metoda nebude vracet žádnou hodnotu. Za typem následuje název metody (`accelerate`, zrychli) a jeden parametr typu `int`. V těle metody zvyšujeme hodnotu atributu `speed`. Všimněte si, že v této metodě nám opět nekoliduje název parametru s atributem, a proto jsme

[2]Mohli bychom místo `private` použít i klíčové slovo `public` a atribut by byl přístupný z jakéhokoliv objektu. To by však narušilo zapouzdření objektu, proto se atributy ve většině případů deklarují jako `private`.

[3]Kdybychom použili klíčové slovo `private`, znamenalo by to, že danou metodu mohou volat pouze metody dané třídy. To se používá v případě, že si chceme vytvořit nějakou pomocnou metodu, u které nechceme, aby ji používal někdo jiný.

nepoužili `this`..

Metoda `brake` (brzdi) je velmi podobná metodě `accelerate`. Zajímavé na ní je, že kromě pouhého snížení rychlosti rovněž zajistí, že se hodnota nedostane do záporných čísel, tj. postará se o to, že objekt je v konzistentním stavu.

Poznámka 3 (K zamyšlení) *Zkuste vymyslet, jak tento objekt dostat do nekonzistentního stavu jiným způsobem.*

Třetí metoda `getSpeed()` slouží k zjistění toho, jak rychle auto jede. Tato metoda nemá žádný parametr a pomocí příkazu `return` vrací hodnotu typu `int`. Poslední metoda `printOut()` slouží k vypsání informací o automobilu.

Poznámka 4 *V jazyce Java jsou zažité konvence pro pojmenování tříd, metod a proměnných, které jsou dodržovány napříč většinou projektů.*

1. *Název třídy by měl obsahovat podstatné jméno. První písmeno je velké a ostatní malé. Pokud se název skládá z více slov, jsou druhé a další slova ,,odděleny" velkými prvními písmeny např.* `Car`, `PersonalComputer`.
2. *Název proměnné, parametru nebo atributu začíná vždy malým písmenem a další písmena jsou malá, např.* `c`, `speed`. *Více slov se opět ,,odděluje" velkými prvními písmeny, např.* `currentSpeed`.
3. *Název metody by měl obsahovat sloveso a platí pro něj stejná pravidla jako pro proměnné, např.* `accelerate`, `getSpeed`.

V názvech se obvykle nepoužívá znak _ (znak podtrženo). Dodržování těchto konvencí je zásadní pro dobré porozumění kódu. Pokud programátor v kódu vidí `Car` *je již odpohledu zřejmé, že se jedná o třídu, pokud vidí* `car` *je zřejmé, že se jedná o proměnnou, attribut nebo parametr obsahující instanci objektu reprezentující auto. Proto je důležité tyto konvence v maximální možné míře dodržovat.*

Nyní již máme třídu `Car` a nastal čas si ji vyzkoušet a vytvořit nové objekty. Vytvoříme si zkušební třídu `CarTest` s jedinou metodou, podobně jako u třídy `MyFirstJavaProgram`.

```
public static void main(String[] args) { }
```

Nové objekty se v Javě vytváří pomocí operátoru `new`, který má tvar:

```
new Trida(parametr1, parametr2, ..., parametrN)
```

`Trida` označuje název třídy a `parametr1`, ..., `parametrN` jsou parametry, které jsou předány konstrukturu, při vytvoření objektu. V našem případě by kód, který vytvoří dva objekty třídy `Car` a přiřadí je do proměnných vypadal následovně.

```
Car blueCar = new Car("1A2 4816", "blue");
Car redCar = new Car("0M1 1235", "red");
```

Všimněte si, že jako typ proměnné je zde použit název třídy. Jinými slovy nám to říká, že proměnná `blueCar` (resp. `redCar`) může obsahovat pouze objekty, které jsou instancí třídy `Car`.

Máme-li objekty, můžeme volat jejich metody. K tomu slouží operátor `.` (tečka), který se používá společně s názvem metody a seznamem hodnot, které jsou metodě předány jako parametry.

```
objekt.metoda(parametr1, ..., parametrN)
```

Úplný kód, který ověří naši třídu, by mohl vypadat takto.

```
public class CarTest {
  public static void main(String[] args) {

    // vytvoří objekty
    Car blueCar = new Car("1A2 4816", "blue");
    Car redCar = new Car("0M1 1235", "red");

    // vypíše informace o modrém autě
    // zavoláním metody printOut()
    blueCar.printOut();

    // analogicky pro červené auto
    redCar.printOut();

    // zavolá metodu accelerate
    // a zrychlí modré auto na 50 km/h
    blueCar.accelerate(50);

    // ověříme rychlost
    blueCar.printOut();
    // zpomalíme o 20 km/h
    blueCar.brake(20);

    // opět ověříme rychlost
    blueCar.printOut();

    // zabrzdíme úplně
    blueCar.brake(100);

    // opět ověříme
    blueCar.printOut();
```

```
  }
}
```

Úkoly k procvičení 1 *Vytvořte třídu* `Rectangle`*, která bude reprezentovat obdélník. Bude mít dva celočíselné atributy odpovídající jednotlivým stranám. Třída bude mít dvě metody: (i)* `int getArea()`*, která vrátí velikost plochy daného obdélníku. (ii)* `void printOut(char vertical, char horizontal)`*, která vykreslí obdélník pomocí znaků* `vertical` *(znak pro svislé úseky) a* `horizontal` *(znak pro vodorovné úseky), podobně jako v minulém semináři.*

Úkoly k procvičení 2 *Vytvořte třídu* `Light` *představující objekty, které bychom mohli v běžném jazyce nazvat světlo (nebo reflektor). Světlo bude mít dva atributy – barvu (textový řetězec) a příznak, zda svítí, či nesvítí. Objekt bude mít dvě metody* `turnOn()`*,* `turnOff()`*, které světlo rozsvítí, resp. zhasnou. A metodu* `printOut()`*, která vypíše stav světla.*

Příklad použití:

```
Light blueLight = new Light("Blue");
blueLight.printOut();      // vypíše []
blueLight.turnOn();
blueLight.printOut();      // vypíše [Blue]
blueLight.turnOff();
blueLight.printOut();      // vypíše []
```

Úkoly k procvičení 3 *Vytvořte třídu* `CheapLight`*, která bude mít stejné metody jako třída* `Light`*, s tím rozdílem, že po deseti přepnutích ze stavu vypnuto do stavu zapnuto se vypíše na textový výstup text „Bang" a světlo již nepůjde zapnout. Demonstrujte tuto funkcionalitu s pomocí cyklů.*

1.1 Práce s objekty

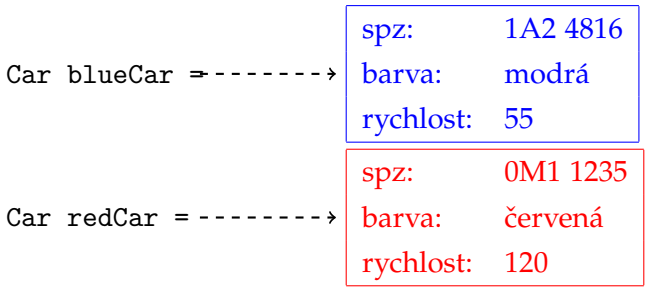
Již víme jak definovat třídy a vytvářet objekty. Nyní se podíváme na některé vlastnosti, které na první pohled nemusí být zřejmé, ale jsou zásadní pro pochopení objektově orientovaného programovaní.

1.1.1 Identita objektů

Ukázali jsem si, jak vytvořit dvě proměnné `redCar` a `blueCar` a uložili do nich dvě instance třídy `Car`.

```
Car blueCar = new Car("1A2 4816", "blue");
Car redCar = new Car("0M1 1235", "red");
```

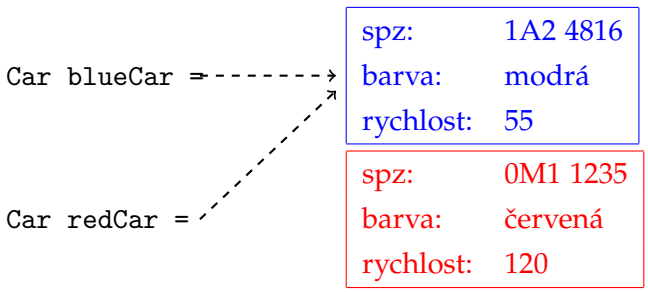
Tyto operace je nutné vnímat ve dvou oddělených krocích. (i) Nejdříve se v paměti vytvoří objekt, instance třídy `Car`. (ii) Do proměnné je uložen *odkaz* na tento objekt. Graficky by se to dalo znázornit:



Co se stane, když provedeme následující přiřazení?

```
redCar = blueCar;
```

Proměnná `redCar` bude také odkazovat na objekt reprezentující modré auto. Graficky si to můžeme představit následovně.



Proměnné `redCar` i `blueCar` nyní odkazují na stejný objekt. Zavoláte-li metodu `redCar.printOut()`, či `blueCar.printOut()`. Výsledek bude identický. To se týká i ostatních metod. Jakékoliv změny v objektu odkazovaném z proměnné `blueCar` se projeví i v objektu odkazovaném z proměnné `redCar` a opačně. Pracujeme vždy se stejným objektem, bez ohledu na to, ze které proměnné k němu přistupujeme. Vyzkoušejte si:

```
blueCar.accelerate(10);
blueCar.getSpeed();    // => 10
redCar.getSpeed();     // => 10

redCar.brake(3);
blueCar.getSpeed();    // => 7
redCar.getSpeed();     // => 7
```

Zda dvě proměnné odkazují či neodkazují na stejný objekt můžeme ověřit pomocí operátorů ==, !=. V našem případě:

```
// pred prirazenim
```

```
(redCar == blueCar)     // false

// po prirazeni
redCar = blueCar
(redCar == blueCar)     // true
```

Je nutné zdůraznit, že operátor == (resp. !=) porovnává pouze to, zda jeho operandy odkazují na *identický* objekt v paměti. Pokud bychom měli dva objekty se shodnými atributy, bude jejich porovnání pomocí == vždy nepravda.

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

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

Poznámka 5 ***(K zamyšlení)****. Jak bychom mohli postupovat, kdybychom potřebovali otestovat, že dva objekty stejné třídy mají shodné hodnoty atributů?*

Náš příklad s přiřazením `redCar = blueCar;` otevírá ještě jednu zajímavou otázku: *Co se stane s objektem „červené auto"?* Stručná odpověď by mohla znít: *To není náš problém.* Tato odpověď, ač poněkud neurvalá, poměrně dobře vystihuje přístup, který uplatňuje jazyk Java při práci s objekty.

V reálném světě, máme-li nějaký objekt a již jej nepoužíváme, měli bychom se postarat o jeho bezpečnou likvidaci. Avšak jazyk Java, aby programátorovi ušetřil práci a předešlo se některým typům chyb, používá automatickou správu paměti, tzv. *garbage collector*, který tuto povinnost bere na sebe. To znamená, že program může vytvářet objekty podle svých potřeb, a pokud již dál nepotřebuje pracovat s nějakým objektem, prostě s ním přestane pracovat. Nic víc dál dělat nemusí. Objekty, s nimiž se přestalo pracovat, nadále zůstávají v paměti. V momentě, kdy množství spotřebované paměti překročí určitou mez, se spustí garbage collector. Ten projde objekty v paměti, identifikuje ty „nepoužívané" a následně je z paměti odstraní. Uvolněná paměť je pak použita pro nové objekty.

Jak ale rozpoznat již nepoužívané objekty? Garbage collector využívá jednoduchou úvahu. Pokud na objekt nevede žádný odkaz z aktuálně prováděného kódu programu, znamená to, že neexistuje způsob, jak se k takovému objektu dostat. Takový objekt je evidentně nepoužívaný a můžeme jej zrušit. Vrátíme-li se k původní otázce, k osudu objektu „červené auto", můžeme říct, že po přiřazení `redCar = blueCar;` bude objekt „červené auto" ještě nějakou dobu v paměti, i když se k němu nebude dát přistupovat, protože na něj nepovede žádný odkaz, a po čase bude tento objekt z paměti automaticky odstraněn.

Předchozí odstavce nám říkají, co se stane s objekty, na které nevede žádný odkaz. Nabízí se ještě související otázka. Můžeme mít proměnnou, která neodkazuje na žádný objekt? Odpověď zní: *Ano, ale měli bychom se tomu vyhnout.* V Javě pro tyto účely máme speciální hodnotu `null`, která značí, že proměnná neodkazuje na žádný objekt. Tato hodnota se typicky používá jako příznak, že objekt, se kterým chceme pracovat, neexistuje. Typicky buď proto, že ještě nebyl vytvořen, nebo protože takový objekt nelze získat. V programech bychom

se ale použití hodnoty `null` měli vyhýbat a pracovat s ní obezřetně. Zavoláme-li metodu objektu, na který nevede platný odkaz, program skončí chybou, tzv. *výjimkou*. Vyzkoušejte si:

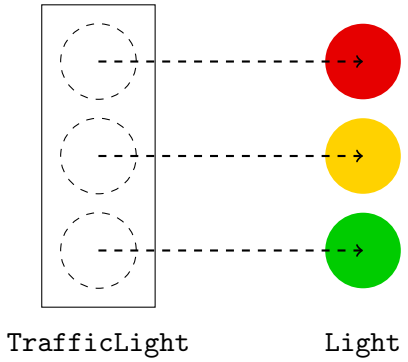
```
Car car = null;
car.printOut();
```

1.1.2 Skládání objektů

Dalším ze základních principů objektového programování je tvorba složitejších objektů z objektů jednodušších. Jelikož jsme si v předchozí kapitole udělali představu o tom, jak se s objekty pracuje, skládání objektů do složitějších celků pro nás bude přímočaré. Stačí si uvědomit, že to, co platí pro proměnné, platí i pro atributy. Jinými slovy, atributy mohou obsahovat odkazy na další objekty. Složitější objekty vznikají tak, že si v atributech udržují odkazy na další objekty.

Jedním z úkolů k procvičení bylo vytvořit třídu `Light` představující světlo (reflektor). Každý objekt třídy `Light` má svou barvu a lze jej rozsvítit nebo zhasnout. My teď tento objekt použijeme a vytvoříme si novou třídu představující semafor pro řízení dopravy.

Naše třída `TrafficLight` má tři atributy (řádky 6 až 8). Každý atribut představuje jedno světlo semaforu (červené, žluté, zelené) a je inicializován v konstruktoru (řádky 10 až 14). Můžeme si to představit tak, že každý objekt třídy `TrafficLight` odkazuje na tři objekty třídy `Light` představující jednotlivá světla semaforu, jak ukazuje Obrázek 2. Poznamenjme, že kdykoliv vytvoříme semafor, vytvoříme pochopitelně i jednotlivé jeho reflektory.



Obrázek 2: Ilustrace třídy `TrafficLight`

Náš semafor je ovládán pomocí tří metod `setStop`, `setCaution` a `setGo`, které fungují tak, že jsou všechna světla nejdřív nastavena na *zhasnuto* a pak je jedno z nich nastaveno na *zapnuto*. Pro vypnutí všech světel jsme si vytvořili pomocnou metodu `turnAllOff` (řádky 42 až 46). Jelikož je to pomocná metoda, je deklarována jako `private`, aby ji jiný objekt nemohl použít. Všimněte si, že v celém kódu nijak nepracujeme s tím, jak se mají jednotlivá světla zapínat, či vypínat, nebo jak je realizován výstup. Pouze předpokládáme, že světlo má metody `turnOn()`, `turnOff()` a `printOut()`.

```java
/**
 * Trida predstavujici semafor pro rizeni dopravy.
 */
public class TrafficLight {

  private Light red;
  private Light yellow;
  private Light green;

  public TrafficLight() {
    red = new Light("red");
    yellow = new Light("yellow");
    green = new Light("green");
  }

  /** nastavi znameni "stop" */
  public void setStop() {
    turnAllOff();
    red.turnOn();
  }

  /** nastavi znameni "pozor" */
  public void setCaution() {
    turnAllOff();
    yellow.turnOn();
  }

  /** nastavi znameni "jed" */
  public void setGo() {
    turnAllOff();
    green.turnOn();
  }

  /** zobrazi stav semaforu */
  public void printOut() {
    red.printOut();
    yellow.printOut();
    green.printOut();
  }

```

```
41    /** vypne vsechna svetla */
42    private void turnAllOff() {
43      red.turnOff();
44      yellow.turnOff();
45      green.turnOff();
46    }
47  }
```

Že semafor funguje dle očekávání, si můžete sami vyzkoušet:

```
TrafficLight trafficLight = new TrafficLight();
trafficLight.setStop();
trafficLight.printOut();
trafficLight.setGo();
trafficLight.printOut();
```

Na třídě `TrafficLight` si všimněte ještě dvou praktických věcí, které sice na fungování programu nemají vliv, ale dělají kód třídy srozumitelnější. Jednak je to samotná struktura kódu. Kód třídy lze rozdělit do čtyř pomyslných částí:

1. deklarace atributů,
2. konstruktor(y),
3. veřejné (`public`) metody,
4. pomocné (`private`) metody.

Je zvykem dodržovat tuto strukturu, jelikož usnadňuje orientaci v kódu třídy.

Pro orientaci v programu jsou užitečné také komentáře. Všimněte si především komentáře ve tvaru:

```
/** popis metody */
```

Jedná se o tzv. Javadoc komentář, což je varianta víceřádkového komentáře s tím rozdílem, že za lomítkem následují dvě hvězdičky. Tyto komentáře se píší před deklarace tříd, metod a atributů a popisují, co jednotlivé části programu znamenají a co dělají. Tyto komentáře mají výsadní postavení, protože vývojová prostředí a další nástroje s nimi počítají a jsou schopny zobrazovat při programovaní nápovědu, co daná třída, či metoda dělá, jak ji použít, atd.

Doporučená literatura

1. The Java™ Tutorials: Getting Started
2. The Java™ Tutorials: Learning the Java Language