Další vlastnosti objektového modelu jazyka Java 3

1 Další vlastnosti objektového modelu

V předchozím semináři jsme si představili základní principy na nichž stojí objektově orientované programovaní v Javě. V praxi ale potřebujeme mít často k dispozici další nástroje a prostředky programovacího jazyka, které nám usnadní programování, ty si ukážeme v tomto a dalších seminářích.

1.1 Řetězce

S jedním typem objektů jsme se setkali již v prvním semináři, aniž bychom si to uvědomovali. Tím typem, resp. třídou, jsou řetězce, které jsou reprezentovány třídou `String`. My jsme však objetky této třídy nevytvářeli pomocí operátoru `new`, ale použili jsme tzv. řetězcové literály[1] a překladač, společně s běhovým prostředím se postaral o vytvoření konkrétní instance objektu. Poznamenejme, že třída má i svůj konstruktor a můžeme řetězce vytvářet jako běžné objekty.

```
String s = new String("abcdef");
```

Třída `String` má celou řadu metod, která umožňují pracovat s řetězcem, např. zjistit jeho délku (metoda `String.length()`), zjistit znak na konkrétní pozici (metoda `String.charAt(int)`), vyhledávat v řetězci (metoda `String.indexOf(...)`), získat podřetězec (`String.substring(int, int)`). Důležitá vlastnost je, že žádná z metod nemění řetězec. Řetězce jsou tedy neměnné, což má dva zásadní důsledky.

Zaprvé, řětězce, které jsou zadány jako literály, často sdílí místo v paměti.[2] Můžeme si to vyzkoušet, i když obecně se na to nedá spolehnout.

```
String s = "abcdef";
String t = "abcdef";
boolean equals = (s == t);  // ==> true
```

Pokud ale řetězec vznikne jiným způsobem, dá se očekávat, že řetězce se stejným textem budou reprezentovány odlišnými objekty.

[1] Literál je textový zápis určité hodnoty přímo ve zdrojovém kódu.

[2] Toto je bezpečné, protože objekty jsou vytvářeny překladačem a běhovým prostředím, a nikdy nebudou změněny.

```
String subS = s.substring(0, 3);     // ==> "abc"
String subT = t.substring(0, 3);     // ==> "abc"
boolean subEquals = (subS == subT);  // ==> false
```

Pokud chceme porovnat dva řetězce, je nutné použít metodu `String.equals`.

Poznámka 1 *Porovnání řetezců pomocí operátoru == patří mezi běžné chyby zejména u programátorů se zkušeností z jiných programovacích jazyků, kde je operátor == přetížen tak, aby vedl na porovnání řetězců, např. C#. Záludnost této chyby spočívá v tom, že při jednoduchých testech snadno unikne, protože se kód na první pohled chová tak, jak by se od něj očekávalo.*

Druhý důsledek toho, že řetězce jsou neměnné, je, že při spojení řetezců je vždy v paměti vytvořen nový řetězec. Například v následujícím kódu je na druhém řádku vytvořen zcela nový objekt reprezentující řetězec `"abcdef"`.

```
String s = "abc";
s += "def";
```

Uvažujme, že bychom chtěli sestavit řetězec, který se bude skládat z čísel od 1 do 10 oddělených mezerou. Mohli bychom to provést následujícím kódem.

```
String numbers = "";
for (int i = 1; i <= 10; i++) {
  numbers += i;
  numbers += " ";
}
```

Toto řešení je zcela funkční, pomineme-li fakt, že na konci řetězce přebývá mezera. To v tento okamžik není podstatné. Důležité je, že toto řešení je velmi neefektivní. Problém tkví v tom, že kdykoliv provedeme spojení dvou řetězců pomocí operátoru +, vznikne nový řetězec a původní řetězec je časem odstraněn garbage collectorem. Podobné je to se spojením řetězce a čísla, kde však překladač nejdříve provede převod číselné hodnoty na řetězec. Vznikne nám tak 19 objektů (řetězců), které jsou sestaveny a obratem „zahozeny".

Abychom mohli efektivně sestavovat řetězce, má jazyk Java třídu `StringBuilder`. Objekt třídy `StringBuilder` v sobě postupně sestavuje řetězec pomocí metody `append` a výsledný řetězec lze získat metodou `toString()`. Náš příklad bychom mohli opravit následovně.

```
StringBuilder numbersBld = new StringBuilder();
for (int i = 1; i <= 10; i++) {
  numbersBld.append(i);
  numbersBld.append(" ");
}
String numbers = numbersBld.toString();
```

Toto řešení bude výrazně efektivnější, protože odpadá opakovaná alokace místa v paměti a neustálé kopírování řetězců.

Poznámka 2 *Ve skutečnosti překladač pro veškerá spojení řetezců pomocí operátoru* `+` *vytvoří objekt třídy* `StringBuilder` *a volá metody* `StringBuilder.append()`.

1.2 Pole

Práce s jednotlivými objekty je sice intuitivní a názorná, avšak v praktických aplikacích je často nutné pracovat s celými soubory dat (objektů). Jedním z prostředků, který jazyk Java pro tento účel nabízí, jsou *pole*. Na pole můžeme nahlížet jako na objekt, který obsahuje soubor hodnot (objektů) stejného typu, k nimž se přistupuje pomocí indexů.

Typ pole má tvar `Typ[]`, tj. `int[]` představuje pole celých čísel, `Car[]` pro změnu odpovídá poli objektů třídy `Car`. Nové pole můžeme vytvořit dvěma způsoby – (i) s pomocí operátoru `new` s tím, že uvedeme velikost pole, tj. kolik prvků má pole obsahovat, (ii) výčtem jednotlivých prvků pole.

```
// pole deseti celých čísel
int[] a = new int[10];

// pole pěti automobilů
Car[] cars = new Car[5];
```

Takto vytvořená pole obsahují prvky inicializované na implicitní hodnoty, tj. `0` v případě čísel a `null` v případě pole objektů. Pokud bychom chtěli vytvořit pole a zároveň jej inicializovat, můžeme použít výčet prvků zapsaný do složených závorek.

```
// pole o trech prvcich
int[] a = { 10, 20, 30 };

// pole dvou automobilu
Car[] cars = { new Car("1A2 4816", "blue"),
               new Car("0M1 1235", "red") };
```

K jednotlivým prvkům pole se přistupuje pomocí indexů zapsaných do hranatých závorek, tj. `a[i]` představuje prvek v poli na pozici `i`. Pozor! Máme-li pole s n prvky, první prvek má index 0 a poslední $n - 1$.

```
a[0] = 1; // ulozi do prvniho prvku hodnotu 1

// do druheho prvku ulozi hodnotu prvniho
// zvysenou o 3
a[1] = a[0] + 3;
```

Procházet prvky pole můžeme dvěma způsoby. Buď můžeme pomocí bežných cyklů procházet jednotlivé indexy a přistupovat pomocí nich k prvkům pole.

```
int[] sudaCisla = new int[10];
for (int i = 0; i < sudaCisla.length; i++)
  sudaCisla[i] = i * 2;
```

Případně můžeme použít cyklus typu `for`-each, který se zapisuje ve tvaru: `for (Typ proměnná : pole) { }`

```
for (int x : sudaCisla) {
  System.out.println(x);
}
```

A můžeme jej číst jako: *Každý prvek pole* `sudaCisla` *typu* `int` *ulož do proměnné* `x` *a proveď tělo cyklu*.

Pole mají ještě jedno zajímavé použití, a tím jsou metody s proměnlivým počtem argumentů.[3] Stačí uvést u posledního argumentu `...` (tři tečky) a tento argument, z pohledu volajícího kódu, bude možné použít opakovaně a z pohledu volané metody se bude jevit jako pole, do kterého byly jednotlivé argumenty uloženy. Ukážeme si to prakticky na následující funkci.

```
/** spocita celkovou uhradu a aplikuje na ni zadanou slevu */
public double calculateTotalPrice(double discount, double... itemPrices) {
  double total = 0.0;
  for (double itemPrice : itemPrices) {
    total += itemPrice;
  }
  return total * (1 - (discount / 100));
}
```

Tato metoda má jeden pevně daný argument `discount` typu `double` a potom akceptuje libovolné množství argumentů typu `double`, které představují ceny jednotlivých položek, například na účtence. Tyto argumenty jsou uvnitř metody dostupné jako argument typu `double[]`.

Úkoly k procvičení 1 *Vyzkoušejte metodu zavolat s různými parametry.*

1.3 Metody a atributy tříd

Doposud jsme uvažovali, že atributy udávají vlastnosti objektů a metody s těmito vlastnostmi pracují. Zejména z praktických důvodů jazyk Java umožňuje přiřadit atributy a metody i jednotlivým třídám, říkáme jim proto *metody tříd* a *atributy tříd*. Od běžných atributů a metod se liší tím, že:

[3] Někdy se používá též označení *variadické metody*.

- při jejich deklaraci je použito klíčové slovo `static` (proto se někdy používá obrat *statická metoda* nebo *statický atribut*),
- jsou sdíleny mezi všemi objekty dané třídy,
- není potřeba mít objekt, abychom k nim mohli přistupovat,
- metody tříd mohou přistupovat pouze k atributům tříd, nikoliv k atributům jednotlivých objektů.

Metody tříd se používají v situacích, kdy máme výpočet, který není spojen s nějakým konkrétním objektem a nepotřebuje pracovat s jeho daty. Dejme tomu, že potřebujeme spočítat hodnotu faktoriálu, což je funkce hojně používaná třeba v kombinatorice. Evidentně zde není vhodný objekt, se kterým by tento výpočet šel spojit, takže si pro tento výpočet můžeme vytvořit metodu třídy.

```
public class MathFuncs {

  public static int fact(int n) {
    if (n == 0) return 1;
    else return n * fact(n - 1);
  }
}
```

Metody tříd se v rámci dané třídy volají stejně jako obyčejné metody. Pokud bychom metodu třídy chtěli volat z jiné třídy, použili bychom nejdříve název třídy a operátor `.` (tečka), tj. `MathFuncs.fact(n)`.

Zajímavým zdrojem podobných funkcí je například třída `Math`, která obsahuje matematické funkce `Math.sin(double)` a `Math.cos(double)` počítající sinus, resp. kosinus daného úhlu, `Math.sqrt(double)` počítající druhou odmocninu, aj.

Úkoly k procvičení 2 *Napište metodu třídy* `void factorials(int n)`*, která vypíše prvních* `n` *hodnot faktoriálu. Tuto metodu otestujte.*

Atributy tříd se dají použít v situacích, kdy chceme zavést nějakou konstantu, např.:

```
public class Physics {
  public static final SPEED_OF_LIGHT = 299792458;
}
```

V rámci třídy `Physics` se k hodnotě `SPEED_OF_LIGHT`[4] přistupuje jako k běžném atributu. Pokud bychom chtěli konstantu použít v jíné třídě (resp. objektu), použili bychom opět název třídy, např. `Physics.SPEED_OF_LIGHT`.

Další využití najdou atributy tříd v momentě, kdy potřebujeme sdílet informaci mezi objekty stejné třídy. Dejme tomu, že bychom chtěli každému objektu přiřadit unikátní hodnotu, třeba automobilům z našeho

[4]Názvy konstant se obvykle píší velkými písmeny a jednotlivá slova jsou oddělena znakem _ (podtržítko).

příkladu přiřadit unikátní sériové číslo. Můžeme to udělat tak, že si nejdříve zavedeme atribut třídy (`unitsProduced`), který nám bude uchovávat počet již vytvořených automobilů. Tento atribut bude sdílen všemi instancemi třídy `Car`. Následně při inicializaci objektu hodnotu tohoto atribut (`unitsProduced`) zvýšíme a přiřadíme jeho hodnotu do atributu nově vytvořeného automobilu. Upravený kód může vypadat následovně.

```
public class Car {

  /** celkovy pocet vytvorenych aut */
  private static int unitsProduced;

  /** nove pridany atribut se seriovym cislem */
  private final int serialNo;

  private final String plateNo;
  private final String color;
  private int speed;

  // konstruktor
  public Car(String plateNo, String color) {
    this.plateNo = plateNo;
    this.color = color;
    this.speed = 0;

    // zvetseni poctu produkovanych aut
    // prirazeni noveho serioveho cisla
    this.serialNo = (++unitsProduced);
  }
  // ... puvodni kod
```

Úkoly k procvičení 3 *Ověřte, že skutečně každý objekt má unikátní sériové číslo.*

Poznámka 3 *Vraťme se ještě k prvnímu programu, se kterým jsme se setkali. Nyní již totiž máme potřebné znalosti k tomu, abychom pochopili, co přesně znamená.*

```
public class MyFirstJavaProgram {
  public static void main(String[] args) {
    System.out.println("Prvni program!"));
  }
}
```

Vytvořili jsme třídu `MyFirstJavaProgram`*, která má jednu metodu (metodu třídy)* `void main(String[])`*. Tato metoda je automaticky zavolána při spuštění programu a jako parametr je jí předán seznam parametrů, se kterými byl program spuštěn. To se typicky používá při spouštení programu z příkazové řádky. Obvykle je toto pole prázdné.* `System` *je třída, která obsahuje metody a atributy pro přístup k systému. Jedním z nich je i atribut třídy* `out`*, který odkazuje na objekt, který nám umožňuje vypisovat informace do textové konzoly, např. pomocí metody* `println`*.*

1.4 Návrhové vzory využívající metody a atributy tříd

Metody a atributy tříd jdou do jisté míry proti principům objektově orientovaného programovaní, ale mají často nezastupitelnou roli. Často se s nimi můžeme potkat v situacích, kdy potřebujeme řídit vytváření objektů. Uvažujme, že v případě některých objektů, potřebujeme mít v programu vždy nejvýše jednu instanci. Takovým objektem může být třeba objekt, který se stará o logování zpráv na terminál nebo do souboru, a který používají různé třídy.

Takový objekt bychom mohli navrhnout následovně.

```
1  public class Logger {
2
3    private static Logger logger;
4
5    /** identifikator zpravy */
6    private long eventId;
7
8    /** soukromy konstruktor */
9    private Logger() {
10     this.eventId = 1;
11   }
12
13   /** vrati instanci loggeru */
14   public static Logger getInstance() {
15     if (logger == null) {
16       logger = new Logger();
17     }
18     return logger;
19   }
20
21   /** vypise na terminal zpravu vcetne jejiho identifikatoru */
22   public void log(String message) {
23     System.out.println(eventId + ": " + message);
24     eventId++;
25   }
```

```
26 }
```

Objekty této třídy mají právě jeden atribut `eventId` identifikující číslo poslední události a metodu `log(String)`, která na výstup programu vypíše číslo události a zadaný text. Dále má tato třída konstruktor, který objekt inicializuje. Na konstruktoru této třídy je zajímavé, že má konstruktor označený jako `private`. To znamená, že žádná jiná třída nemůže tento konstruktor použít. Jak ale v takovém případě vytvoříme instanci této třídy? K tomu slouží metoda, `getInstance()`, která se podívá, jestli v atributu třídy je již nějaká vytvořená instance. Pokud ne, vytvoří ji a uloží ji do attributu třídy (statického atributu) `logger`. V každém případě se vrátí hodnota atributu `logger`.

Použití této třídy by mohlo vypadat následovně:

```
Logger.getInstance().log("Událost #1");
```

Díky tomu, že konstruktor třídy je skryt, nikdo jiný než třída `Logger` nemůže vytvořit svou instanci. Současně díky logice metody `getInstance()` máme garantované, že objekt `Logger` bude vytvořen nanejvýš jednou.

Poznámka 4 *To, co jsem si nyní ukázali, je tzv. návrhový vzor Singleton. Návrhové vzory jsou zaužívaná řešení určitých problémů, se kterými se při programování či návrhu aplikace můžeme setkat. Námi představený návrhový vzor Singleton se používá všude tam, kde chceme mít zajištěnu existenci maximálně jedné instance dané třídy.*