Dobré a špatné implementační praktiky

Obsah

(Ne)měnnost objektů

String

- `String` je samozřejmě typ neměnný (immutable). Jakmile je jednou vytvořen a naplněn hodnotou, hodnota tam stále zůstává. Jsou pro to dobré důvody, např. lze tyto objekty pohodlně a bez starostí sdílet přes více metod či vláken.
- Operace, která by obsah změnila - např. `toUpperCase`, vrátí odkaz na nový objekt řetězce.
- Modifikovatelnými řetězci jsou `StringBuilder` a `StringBuffer`.

Objektová zapouzdření primitivních typů

- Typy jako `Double, Float, Integer, Character, Long` atd. jsou objektovými reprezentacemi (zapouzdřeními) odpovídajících primitivních typů.

- Lze je tedy použít tam, kde je očekáván objekt, např. v kolekcích.
- Tyto objekty jsou podobně jako `String` neměnné (immutable).

java.util.Date

- `Date` je často používaný a bohužel *modifikovatelný* objekt. Má metodu `setTime`, kterou obsah data změníme.
- Většina aplikací s tím ale nepočítá, datum jednou předané do metody si neduplikuje, ale spoléhá, že se nemění. Proto jsou praktiky používající `setTime` silně nevhodné.

Výjimky

Nemá smysl zachytávat výjimku za každou cenu

Níže uvedený úryvek kódu se snaží zabránit tomu, aby výjimka pronikla ("utekla") z metody main. Způsob, jakým je to uděláno, ale nic neřeší. jen se vypíše chybová hláška z výjimky - a navíc je otázka, zda se má v takovém případě vypsat dosud načtená část seznamu (záleží na požadavcích zadání):

```
public static void main(String[] args) {
   ...
   try{
      BufferedReader input = new BufferedReader(new FileReader(f));
      String line = new String(input.readLine());
      while (line!=null){
         ...
      }
      ...
   } catch (IOException e) {
      System.out.println(e);
   }
   p = employee.first();
   while (p!=null){
      System.out.println(p);
      p = employee.higher(p);
   }
}
```

Pokud zadání explicitně nestanoví, že se mají vypsat aspoň správně načtené záznamy, je lepší prostě výjimku propustit z main:

```
public static void main(String[] args) throws IOException {
   ...
```

```
   BufferedReader input = new BufferedReader(new FileReader(f));
   String line = new String(input.readLine());
   while (line!=null){
      ...
   }
   p = employee.first();
   while (p!=null){
      System.out.println(p);
      p = employee.higher(p);
   }
}
```

Když už se zachytí, musí se na ni reagovat - SMYSLUPLNĚ reagovat.

Kdy použít výjimku?

Výjimka je obecně technikou, jak zachytit a reagovat na *mimořádnou* událost v programu. Zachycení výjimky by nemělo nahrazovat jednoduchý test. Občas je to ale vidět (sic!):

```
public void writeInfo(Person p) {
   try {
      System.out.print("Info about a Person: ");
      System.out.println(p.getInfo);
   } catch(NullPointerException npe) {
      System.out.println("Error: No person given");
   }
}
```

Proč ne raději takhle:

```
public void writeInfo(Person p) {
   if(p == null) {
      System.out.println("Error: No person given");
   } else {
      System.out.print("Info about a Person: ");
      System.out.println(p.getInfo);
   }
}
```

Kód je dokonce kratší a hlavně čistší. Všimněte si také podmínky - je psána tzv. v *aserci* (bez negace). Je to čitelnější než:

```
public void writeInfo(Person p) {
   if(p != null) {
      System.out.print("Info about a Person: ");
      System.out.println(p.getInfo);
   } else {
```

```
            System.out.println("Error: No person given");
        }
    }
```

Jaký typ výjimky?

Java na rozdíl od řady jiných OO jazyků rozlišuje výjimky *hlídané* (checked) a nehlídané (běhové, unchecked).

Porovnávání řetězců

Equals nebo ==?

Velmi častou chybou začátečníka, bohužel navíc podpořenou špatnými vzory z jiných jazyků či prostředí, je použití operátoru == k porovnání řetězců na shodu po znacích. Chceme-li zjistit, zda dva řetězce jsou obsahově stejné, tj. stejně dlouhé a mají na stejných pozicích stejné znaky, používáme volání metody equals, která je pro třídu String přetížená tak, aby se dosáhlo požadovaného chování.

Equals není však ve všech případech ideální. Z níže uvedených variant [http://www.odi.ch/prog/design/newbies.php#2] není ideální žádná. Proč?

if (name.compareTo("John") == 0) je trochu zbytečný "kanón na vrabce". Navíc selže s výjimkou, je-li `name == null`.

if (name == "John") je asi většinou špatně, testuje totiž, zda `name` ukazuje na identický řetězec `"John"`. Bohužel to často i "funguje", např. když program vypadá takto:

```
String name = "John";
...
if (name == "John") {
   // opravdu sem dojde, protože Java si oba výskyty lite
   // tj. nahradí jedinou instancí
   System.out.println ("It is really John");
}
```

Ale když jeden z řetězců zkonstruujeme (substring, zřetězení), fungovat to nebude:

```
String origNameFirst = "John ";
String origName = origNameFirst + "Paul Willard";
String name = origName.substring(0, 4);
if (name == "John") {
   // nedojde sem
   System.out.println ("It is John");
}
```

if (name.equals("John"))	už vypadá dobře, ale selže s výjimkou, je-li `name == null`. A nač to testovat zvlášť, když lze napsat rovnou `if ("John".equals(name))`, které tento neduh nemá.
if ("John".equals(name))	rozumné řešení, robustní i proti `name == null` -- pak prostě vrátí `false`.

Viz také Java Quick Reference (Operators and Assignments) [http://www.janeg.ca/scjp/oper/string.html].

Test prázdnosti řetězce

A propos, jde nám o test, zda řetězec existuje, ale neobsahuje žádný znak (jako `""`), nebo může být i null?

if ("".equals(name))	vypadá OK, samozřejmě vrátí false, je-li `name == null`. Je však zbytečně neefektivní, protože volání equals je náročnější, než zjištení deélky řetězce:
if(name.length()==0)	ale to zase selže při `name == null`...

Zřetězování

Klasika - jak spojovat více řetězců

Začátečníka láká použít pro spojování řetězců operátor +, a to i v případě, že se za existující řetězec připojují další znaky či řetězce nebo se zřetězení dokonce opakuje v cyklu.

Co však zřetězení `s = s + " novy";` udělá?

1. Má `s` jako odkaz na objekt s původním řetězcem, `" novy"` jako literál / hotový řetězec v paměti.
2. Vytvoří nový řetězec jako prázdný `StringBuilder`,
3. ty původní (`s` a `" novy"`) do něj přikopíruje.
4. Převede `StringBuilder` na `String`.
5. Původní objekt `s` se "zapomene".

A toto provádět v cyklu? To je trochu moc operací, ne!?

První příklad na Java Anti-patterns [http://www.odi.ch/prog/design/newbies.php#0] také ukazuje tuto extrémně neefektivní konstrukci.

Něco jako `s += " novy"` sice vypadá hezky, ale v tomto ohledu vůbec nepomůže. Jediným rozumným řešením je v cyklu použití StringBuilderu (příp. StringBufferu):

```
StringBuilder sb = new StringBuilder(persons.size() * 16); // well estimated buffe
for (Person p : persons) {
    sb.append(p.getName);
}
```

StringBuilder nebo StringBuffer?

- Obě třídy představují *modifikovatelné* řetězce.
- *Normální `String` je immutable, nemodifikovatelný a každá změna - např. pripojení nebo odebrání znaku/ů - znamená konstrukci řetězce nového.*
- Historicky starší je `StringBuffer`, který je synchronizovaný, tzn. může se nad jedním pracovat z více vláken, aniž by se porušila atomicita změnových operací (nedostane se do inkonzistentního stavu).
- Pokud toto nepotřebujeme, resp. pohlídáme si přístup jinak, je lepší použít rychlejší `StringBuilder`.

Tipy pro StringBuilder

Snažme se využívat výhod metod append - např. toho, že existují přetížené varianty pro různé datové typy. Než tohle:

```
sb.append(" age=" + age);
```

které nejprve neefektivně zřetězí String a číslo převedené na String, tak raději:

```
sb.append(" age=");
sb.append(age);
```

Konverze

Z řetězce na číslo či jiný typ - metody *parseXXX*

Rozumíme tím převod znakové reprezentace určité hodnoty na hodnotu *typu* číslo, příp. jiného:

```
String myNumberInString = "-123.456";
double myNumber = Double.parseDouble(myNumberInString); // převede správně na doub
```

Obdobně postupujeme u cílových typů float, byte, short, int, long a boolean.

Přes objektové reprezentace

Variantně lze použít i konstruktory objektových "reprezentací":

```
String myNumberInString = "-123.456";
double myNumber = new Double(myNumberInString); // převede správně na double přes
System.out.println ("test4 -- " + myNumber);
```

Převod na řetězec zřetězením

I pouhý výpis přes System.out.println vede k takové konverzi (a často následnému zřetězení):

```
double x = -123.456;
System.out.println ("test5 -- " + x); // převede x na String a zřetězí
```

Tohle je už ale zbytečné:

```
double x = -123.456;
String s = "" + x; // převede x na String a přiřetězí k prázdnému
```

Explicitní převod na String

Vhodná náhrada předchozího:

```
double x = -123.456;
String s = String.valueOf(x); // převede x na String, nic neřetězí
```