

# Neměnné objekty a záznamy (record)

```
<link rel="stylesheet" href="http://cdnjs.cloudflare.com/ajax/libs/font-awesome/3.1.0/css/font-awesome.min.css">
```

## Neměnné objekty

- Neměnný (*immutable*) objekt nemůže být po jeho vytvoření modifikován
- Bezpečně víme, co v něm až do konce života bude
- Tudíž může být souběžně používán z více míst, aniž by hrozily nekonzistence
- Jsou prostředkem, jak psát robustní a bezpečný kód
- K jejich vytvoření nepotřebujeme žádný speciální nástroj
- Od nových verzí Javy (14+) lze pro tento účel s výhodou využít typy **record** (záznam)

## Příklad neměnného objektu

```
public class Vertex1D {
    private int x;
    public Vertex1D(int x) {
        this.x = x;
    }
    public int getX() {
        return x;
    }
}
...
Vertex1D v = new Vertex1D(1);
// x of value 1 cannot be changed anymore
```

## Totéž jako záznam (**record**)

- Namísto třídy **class** můžeme použít **record**
- Tím se definuje třída, kde každý objekt bude mít vlastnosti **name** a **address**, které se nastaví jednou a už nejdou změnit, čili jako výše **Vertex1D**.

```
public record Person (String name, String address) {}  
//...  
Person p = new Person("Pavel Holec", "Lipová 3, Brno");
```

## Co nabízí záznam?

- Překladač pro nás automaticky pro tuto třídu vytvoří:
  - konstruktor mající parametr pro nastavení každého atributu (hodnoty) v objektu
  - metodu `equals` pro porovnání objektů: dva záznamy budou stejné  $\Leftrightarrow$  jsou stejné příslušné atributy
  - přístupové metody pro čtení atributů, např. `person.name()`
  - "inteligentní" metodu `toString` vracející např. `Person[name=John Doe, address=100 Linda Ln.]`



Jména metod jsou odlišná od konvence používané u JavaBeans nebo obecně u javových objektů: tradičně by bylo `person.getName()`, ale u záznamu `person.name()`

## Výhody a nevýhody neměnných objektů

### Výhody

- je to vláknově bezpečné (*thread safe*) — objekt může být *bezpečně* používán více vlákny naráz
- programátor má jistotu, že se mu obsah objektu *nezmění* — silný předpoklad
- kód je *čitelnější, udržovanější i bezpečnější* (např. útočník nemůže změnit náš token)

### Nevýhody

- chceme-li objekt být jen drobně změnit, musíme vytvořit nový
- to stojí čas a paměť

## Porovnání

- Neměnný objekt vs. konstanta
  - konstanta je jenom jedna (je statická)
  - neměnných objektů může být víc s různými hodnotami
  - kdyby jich bylo více a měly stejnou hodnotu, postrádá smysl
- Neměnný objekt vs. `final`
  - `final` prakticky zakáže změnu odkazu
  - nelze tedy do dané proměnné přiřadit odkaz na jiný objekt
  - samotný objekt ale může být dál "vevnitř" modifikován

## Příklad `final`

```
final Person p = new Person("Marek");
// p = new Person("Jan"); // cannot be done
p.setName("Haha I changed it"); // this works!
```

## Vestavěné neměnné třídy

- Neměnnou třídou v Javě je `String`.
- Má to řadu dobrých důvodů - tytéž jednou definované řetězce lze používat souběžně z více míst programu
- Nicméně i negativní stránky - někdy větší režie spojená s nemodifikovatelností:

```
String s = "Hello " + "World";
```

- Kód vytvoří 3 objekty: "Hello ", "World" a "Hello World".
- Cože? To je tak neefektivní?
- Pokud by vysloveně vadilo, lze místo `String` použít `StringBuilder`, který je modifikovatelný (mutable), viz dále

## String pod lupou

- Podívejme se na rozdíl "Hello" a `new String("Hello")`.

```
new String("Hello")
```

vytvoří pokaždé nový objekt

```
"Hello"
```

funguje na principu: *jestli taký objekt zatím neexistuje, tak ho vytvořím, jinak vrátím odkaz na již existující objekt (uložen v paměti String constant pool)*

```
String s1 = "Hello";
String s2 = "Hello";
boolean bTrue = (s1 == s2); // true, identical objects

s1 = new String("Hello");
s2 = new String("Hello");
boolean bFalse = (s1 == s2); // false, different objects though with same value
```

# Metody třídy `String`

- `char charAt(int index)` — vrátí prvek na daném indexu
- `static String format(String format, Object... args)` — stejné jako `printf` v C
- `boolean isEmpty()` — vrátí `true` jestli je prázdný
- `int length()` — velikost řetězce
- `matches, split, indexOf, startsWith ...`



Více metod najdete v dokumentaci [třídy `String`](#).

- Doporučujeme javadoc prostudovat, používáním existujících metod jsi ušetříte spoustu práce!

## Víceřádkové řetězcové literály = textové bloky

- V nových verzích Javy (15+) je možnost použít speciální formy řetězcových literálů:

```
String s1 = """
    toto je první slovo řetězce, druhé,
    druhý řádek,
    třetí""";
```

- Překladač vytvoří řetězec včetně oddělovačů řádku (dříve bylo možné pomocí znaku `\n`).
- Vynechá počáteční mezery (leading spaces), takže skutečně slovo "toto" bude prvními znaky řetězce `s`.

## Třída `StringBuilder`

```
StringBuilder builder = new StringBuilder("Hello ");
builder.append("cruel ").append("world"); // method chain
builder.append("!");
String result = builder.toString();
```

- `StringBuilder` se průběžně modifikuje, přidáváme do něj další znaky
- Na závěr vytvoříme výsledný řetězec
- `StringBuilder` není *thread safe*, proto existuje její varianta `StringBuffer`.

## Návrhový vzor *Builder*

Třída `StringBuilder` odpovídá návrhovému vzoru *Builder*

- Builder obecně umožňuje konstruovat složitý objekt po částech a až poté získat celý výsledek
- Metody `append` slouží pro přidávání kousků textu, tj. odpovídají metodě `buildPart()` ze vzoru.
- Metoda `toString` slouží k získání výsledného textu, tj. odpovídá metodě `getResult()` ze vzoru.
- `Director` je náš kód používající `StringBuilder`

## Objektové obálky nad primitivními hodnotami

- Java má primitivní typy — `int`, `char`, `double`, ...
- Ke každému primitivnímu typu existuje varianta objektového typu — `Integer`, `Character`, `Double`, ...
- Tyto objektové typy se nazývají *wrappers*.
- Objekty jsou neměnné
- Při vytváření takových objektů není nutné používat `new`,
  - využije se tzv. *autoboxing*, např. `Integer five = 5;`
  - obdobně *autounboxing*, `int alsoFive = five;`

```
Integer objectInt = new Integer(42);
Integer objectInt2 = 42;
```

## Konstanty objektových obálek

- Wrappers (např. `Double`) mají různé konstanty:
  - `MIN_VALUE` je minimální hodnota jakou může `double` obsahovat
  - `POSITIVE_INFINITY` reprezentuje konstantu kladné nekonečno
  - `NaN` je zkratkou Not-a-Number — dostaneme ji např. dělením nuly
- Protože konstanty jsou statické, jejich hodnoty získáme přes název třídy:

```
double d = Double.MIN_VALUE;
d = Double.NEGATIVE_INFINITY;
```

## Metody objektových obálek

- např. pro `Double` existuje `static double parseDouble(String s)` — udělá ze `String` číslo, z `"1.0"`, vytvoří číslo `1.0`
- obdobně pro `Integer` a další číselné typy
- pro převody na číselné typy dále `int intValue()` převod `double` do typu `int`

- `boolean isNaN()` — test, jestli není hodnotou číslo



Více konstant a metod popisuje [javadoc](#).

## Víc hodnot

- Test: Objektové typy (`Integer`) mají od primitivních (`int`) jednu hodnotu navíc — uhádněte jakou!
- je to `null`
- `Integer` je objektový typ, proměnná je odkaz na objekt

## Rozdíl od primitivních typů

Proč tedy vůbec používat primitivní typy, když máme typy objektové?

`int i = 1`

- zabere v paměti právě jen 32 bitů
- používáme přímo danou paměť, jednička je uložena přímo v `i`

`Integer i = 1`

- je třeba alokace paměti pro objekt, zkonstruování objektu s obsahem `1`
- v proměnné `i` je pouze odkaz, je to (nepatrně) pomalejší
- Výkon může být u velkého počtu objektů problém, např. vytvoření milionu proměnných typu `Integer` namísto `int` může mít dopad na výkon a zcela jistě zabere dost paměti, asi zbytečně

## Doporučení

- Používejte *hlavně primitivní typy*
- Využívejte metody objektových typů, hlavně statické, kde není třeba mít objekt
- Řada objektových jazyků vůbec primitivní typy jako v Javě nemá, vše jsou objekty

## Konverze mezi primitivními a objektovými typy

- Java podporuje automatické balení (boxing) a vybalení (unboxing) mezi primitivními typy a wrappery.
- Proto je následující kód je naprosto v pořádku

```
int primitiveInt = 42;
// jakoby new Integer(primitiveInt):
Integer objectInt = primitiveInt;
primitiveInt = new Integer(43);
```

- Jak už bylo řečeno, použití *primitivního typu* je obvykle lepší nápad - jsme v Javě :-)

## Zvláštnosti

- Zajímavost, anebo spíš podraz Javy:

```
Integer i1 = 100; // between -127 and 128
Integer i2 = 100;
boolean referencesAreEqual = (i1 == i2); // true

i1 = 300;
i2 = 300;
boolean referencesNotEqual = (i1 == i2); // false
```

- Poučení: objekty pomocí `==` obvykle neporovnáváme (budeme se učit o `equals`).

## Proč tak podivné chování?

- Optimalizace využití paměti: *"valueOf() returns an Integer instance representing the specified int value. This method will always cache values in the range -128 to 127, inclusive, and may cache other values outside of this range."*
- Důsledek použití návrhové vzoru *Flyweight (muší váha)*, kdy konstruktor (respektive autoboxing kód v Javě) někdy vrací již dříve vytvořenou instanci (de facto *Singleton*), jindy zas zcela novou instanci.