# Základní pojmy OOP. Třída, objekt, jeho vlastnosti. Metody, proměnné. Konstruktory.

August 28, 2014

## 0.1 Úvod do objektového programování (1)

- Pojmy třída, objekt
- Deklarace a definice tříd, jejich vlastnosti (proměnné, metody)
- Vytváření objektů, proměnné odkazující na objekt
- Jmenné konvence – jak tvořit jména tříd, proměnných, metod

## 0.2 Úvod do objektového programování (2)

- Použití objektů, volání metod, přístupy k proměnným
- Modifikátory viditelnosti (public, private...)
- Konstruktory (dotvoří/naplní prázdný objekt)
- Přetěžování metod (dvě metody se stejným názvem a jinými parametry)

# 1 Třída, objekt, jejich vlastnosti

## 1.1 Co je třída a objekt?

**Třída** (také poněkud nepřesně zvaná *objektový typ*) představuje skupinu objektů, které nesou stejné vlastnosti, přičemž "stejné" je myšleno *kvalitativně, typově*, nikoli ve smyslu konkrétních hodnot (ty jsou pro různé objekty téže třídy různé).

- Např. všechny objekty třídy `Person` mají vlastnost `name`,
- tato vlastnost má však obecně pro různé lidi různé hodnoty – lidi mají různá jména

## 1.2 Příklad

*Objekt* je jeden konkrétní jedinec (instance, reprezentant či entita) příslušné třídy.Pro konkrétní objekt *nabývají vlastnosti* deklarované třídou *konkrétních hodnot*Příklad:

- Třída `Person` má vlastnost `name`

- Objekt `panProfesor` typu `Person` má vlastnost `name` s hodnotou `"Václav Klaus"`.

### 1.3 Vlastnosti objektu (hlavní)

- *proměnné* neboli *atributy*
- *metody*

Vlastnosti objektů – proměnné/atributy i metody – je třeba *deklarovat*.viz Oracle The Java(TM) Tutorials Lesson: Classes and Inheritance (`http://docs.oracle.com/javase/tutorial/java/javaOO/classes.html`)

### 1.4 Vlastnosti objektu (2)

Proměnné/atributy

1. jsou nositeli "pasivních" vlastností, jistých *charakteristik* objektů
2. de facto jde o datové hodnoty vložené (zapouzdřené, přebývající) v objektu

Metody

1. jsou nositeli "výkonných" vlastností; "schopností" objektů
2. de facto jde o funkce/procedury pracující (převážně) nad proměnnými objektu

## 2 Deklarace a použití třídy

### 2.1 Příklad - deklarace třídy Person

- deklarujme třídu objektů – lidí

```
public class Person {
   private String name;
   private int age;
   public Person(String name, int age) {
   // pomocí this odlišíme proměnnou objektu od parametru!
      this.name = name;
      this.age = age;
   }
   public void writeInfo() {
      System.out.println("Person:");
      System.out.println("Name " + name);
      System.out.println("Age  " + age);
   }
}
```

## 2.2 Příklad použití třídy Person

Použijeme ji v programu, tzn.:

1. Vytvoříme *instanci* (objekt) typu `Person`.
2. Vypíšeme informace o něm pomocí *metody* tohoto objektu.

## 2.3 Příklad použití třídy Person v programu

Mějme deklarovánu třídu `Person`. Metoda *main* v následujícím programu `Demo`:

1. deklaruje dvě lokální proměnné typu Person - budou obsahovat odkazy na následně vytvořené objekty - lidi
2. vytvoří tyto dva lidi (pomocí `new Person`)
3. zavolá jejich metody `writeInfo()`

## 2.4 Příklad použití třídy Person (dvě instance)

```
public class Demo {
   public static void main(String[] args) {
      Person ales = new Person("Ales Necas", 38);
      Person beata = new Person("Beata Novakova", 36);
      ales.writeInfo();
      beata.writeInfo();
   }
}
```

Tedy: vypíší se informace o obou vytvořených objektech – lidech.Nyní podrobněji k *proměnným* objektů.

## 2.5 Příklad použití třídy Person (2)

Ve výše uvedeném programu znamenalo na řádku:

```
Person ales = new Person("Ales Necas", 38);
```

Person ales je pouze deklarace (tj. určení typu) proměnné *ales* - bude typu *Person*.Až ales = new Person ("Ales Necas", 38) je samotným vytvořením objektu osoby (*Person*) se jménem `Ales Necas` a přiřazení odkazu na něj do proměnné `ales`.Lze napsat zvlášť do dvou řádků nebo (tak jak jsme to udělali) na řádek jeden.Každý příkaz i samostatně stojící deklaraci ukončujeme středníkem.

## 2.6 Vytváření objektů

Ve výše uvedených příkladech jsme objekty vytvářeli voláním new Person(...) a bezděčně jsme tak použili

- **operátor new**, který *vytvoří prázdný objekt typu Person* a vzápětí sám provede
- volání **konstruktoru**, který prázdný objekt *naplní počátečními údaji* (daty).

### 2.7 Shrnutí

Objekty

- jsou instancemi "své" třídy
- vytváříme je operátorem new – voláním konstruktoru
- vytvořené objekty ukládáme do proměnné stejného typu (nebo typu předka či implementovaného rozhraní – o tom až později)
- Pozn. Ve skutečnosti se v Javě nikdy celé objekty do proměnné neukládají, jde vždy o uložení pouze odkazu (adresy) na objekt.

## 3 Atributy (proměnné)

### 3.1 Atributy (proměnné)

Terminologie: pro data zapouzdřená do objektů budeme používat záměnným způsobem označení atributy či proměnné. Je to totéž. Atribut je jednoznačnější, jde vždy o datovou položku objektu či třídy, zatímco proměnné mohou být i lokální ("pomocné") uvnitř metody.

- Položky `name` a `age` v předchozím příkladu jsou **atributy/proměnné** objektu `Person`. Jsou deklarovány v těle deklarace třídy *Person*.
- Deklarace atributu/proměnné objektu má tvar: modifikátory TypProměnné jménoProměnné; např. private int age;

### 3.2 Datové typy primitivní

- Výše uvedený atribut/proměnná `age` měl/a datový typ *int* (32bitové celé číslo).
- Tedy: proměnná takového typu nese *jednu hodnotu typu celé číslo* (v rozsahu $-2^{31}..2^{31}-1$);
- Kromě celých čísel *int* nabízí Java celou řadu dalších *primitivních* datových typů (pro celá i necelá čísla, logické hodnoty, znaky).
- Primitivní (základní, dále nedělitelné) typy jsou Javou dané napevno, programátor je nedefinuje ani nemodifikuje (na rozdíl např. od C/C++), nýbrž jen používá.
- V Javě tedy neexistuje (na rozdíl od C/C++) možnost typy modifikovat (např. unsigned int).

### 3.3 Datový typ objektový

Tam, kde nestačí jednoduché hodnoty (tj. primitivní typy), musíme použít typy *složené*, **objektové**.

- Objektovými typy v Javě jsou **třídy** (class) a **rozhraní** (interface) i **pole**. Třídy už jsme viděli na příkladu *Person*.

- Existují třídy definované přímo v Javě, v knihovně Java Core API.
- Nenajdeme-li třídu, kterou potřebujeme, v Java Core API ani v nám dostupných a použitelných knihovnách, můžeme si ji nadefinovat sami.

# 4 Konvence pro psaní kódu

## 4.1 Proč konvence

- Konvencemi rozumíme zaužívaná doporučení, která především:
- Usnadňují čtení cizího (i vlastního) kódu.
- Eliminují tím chybovost.
- Zvyšují produktivitu, šetří čas vlastní i ostatních.
- Dodržování konvencí nehlídá překladač: i kód nedodržující konvence může být přeložitelný a funkční. Hlídáme si to sami tím, že to dle konvencí vědomě píšeme + můžeme použít specializované nástroje pro kontrolu.

## 4.2 Konvence pro psaní kódu

- Doporučení přímo od Sun/OracleCode Conventions for the Java Programming Language (`http://www.oracle.com/technetwork/java/index-135089.html`)
- Konvence užívané knize Cay Horstmann: Big Java (`http://horstmann.com/bigj/style.html`) popsané v kompaktní formě na 99 % shodné s naším pohledem na věc.
- Do námi užívaného prostředí BlueJ je možné doinstalovat rozšíření pro hlídání stylu kódování BlueJ Checkstyle Extension (`http://bluejcheckstyle.sourceforge.net/`), které hlídá dvě výše uvedené skupiny konvencí

## 4.3 Jmenné konvence především pro proměnné

- týkají se jak lokálních proměnných v metodách, tak atributů
- netýkají se statických atributů a hlavně ne konstant
- jména proměnných začínají malým písmenem
- nepoužíváme diakritiku (problémy s editory, přenositelností a kódováním znaků) – a to přesto, že Java ji i v identifikátorech povoluje
- raději ani český/slovenský či jiný národní jazyk, angličtině rozumí více lidí
- je-li to složenina více slov, pak je *nespojujeme podtržítkem*, ale další začne velkým písmenem (tzv. "CamelCase", v případě proměnných tedy přesněji "camelCase")

### 4.4 Jmenné konvence – příklady atributů

- `private int yearOfBirth;` je identifikátor se správně (vhodně) utvořeným jménem, zatímco:
- `private int YearOfBirth;` není vhodný identifikátor proměnné v Javě (začíná velkým písmenem)
- `private int rokNarozeni;` rovněž není vhodný identifikátor proměnné v Javě (zahraniční kolegové nebudou bez dalšího komentáře rozumět, co v té proměnné je)

### 4.5 Jmenné konvence – závěrem

- **Dodržování jmenných konvencí je základem psaní srozumitelných programů a bude vyžadováno, sledováno a hodnoceno v odevzdávaných úlohách i písemkách.**
- **Konvence se bohužel liší od zvyklostí v příbuzných jazycích.**
- V Javě se např. nepoužívá v názvech tříd znak *podtržítka* – je to teoreticky možné, ale skoro nikdo tak nečiní.
- Poměrně málo často se v názvech tříd či proměnných používají číslice. Občas ano, ale nikoli na začátku a nejspíše tam, kde jde o zvláštní konkrétní význam daného čísla, např. Counter32bit.
- Rovněž se v názvech proměnných *nepoužívá* tzv. "maďarská notace", kde se připojují předpony např. dle *datového typu* proměnné.

## 5 Použití atributů objektů

### 5.1 Zápis přístupu k atributům

Atributy objektu odkazujeme pomocí "tečkové notace":

```
public class Demo2 {
   public static void main(String[] args) {
      // vytvoření objektu ...
      Person ales = new Person("Ales Necas", 38);
      // přístup k (čtení) jeho proměnné ...
      System.out.println(ales.name);
      // modifikace (zápis do) jeho proměnné
      ales.name = "Aleš Novák";
   }
}
```

*V reálu ale tento obrat uvidíme zřídka, protože k proměnným v objektech raději přistupujeme pomocí metod, např. get/set*

## 5.2 Atributy – modifikátory viditelnosti

Viditelnost, tzn. přímá přístupnost atributů (i metod) může být řízena uvedením tzv. *modifikátorů* před deklaraci prvku, viz výše:

```
// public = viditelnost odevšad:
public class Person {

// private = viditelnost pouze zevnitř této třídy:
private String name;
```

*Modifikátorů* je více typů (ještě další dva), zdaleka nejběžnější jsou dva právě zmíněné: private a public.

## 5.3 Čtení hodnot atributů

Objektů (tzv. *instancí*) stejného typu (tj. stejné třídy) si můžeme postupně vytvořit více:

```
// vytvoření prvniho objektu
Person ales = new Person("Ales Necas", 38);
// vytvoření druheho objektu ...
Person petr = new Person("Petr Svoboda", 36);

// přístup k (čtení) proměnné prvního objektu
System.out.println(ales.name);
// přístup k (čtení) proměnné druhého objektu
System.out.println(petr.name);
```

Existují tady dva objekty, každý má své (obecně různé) hodnoty proměnných – např. jsou různá jména obou lidí.

# 6 Metody – definice, volání, návrat

## 6.1 Co a k čemu je metoda?

Nad *existujícími* (vytvořenými) objekty můžeme volat jejich *metody*. Metoda je:

- podprogram (v terminologii neobjektových jazyků – funkce, procedura), který *primárně pracuje s proměnnými "mateřského" objektu*,
- může mít další *parametry*,
- může ve svém kódu (těle) deklarovat *lokální proměnné* – v našem příkladu metoda `main` deklarovala proměnné `ales`, `petr`.

## 6.2 Atribut vs. lokální proměnná

Pro výsledky a mezivýsledky výpočtů používáme na ukládání hodnot lokální proměnné nebo atributy objektů. Rozdíl mezi lokální proměnnou a atributem (proměnnou objektu) je značný:

- Hodnota uložená v atribut objektu je "trvalá" ve smyslu, že přetrvává (až do přiřazení jiné) po celou dobu existence daného objektu.
- U lokální proměnné v metodě platnost skončí (zruší se) tato proměnná ukončením dané metody.

Metoda může *vracet hodnotu* podobně jako v Pascalu *funkce*.

## 6.3 Metody – deklarace

Každá metoda se musí ve své třídě *deklarovat*. V Javě *neexistují metody deklarované mimo třídy* (tj. Java nezná žádné "globální" metody).

## 6.4 Metody – příklad

Výše uvedená třída `Person` měla metodu na výpis informací o daném objektu/člověku:

```
public class Person {
   private String name;
   private int age;
   public Person(String name, int age) {
      this.name = name;
      this.age = age;
   }
   //... zde jsou další metody
}
```

## 6.5 Metody – použití

Výše uvedená třída `Person` měla metodu pro výpis informací o daném objektu/člověku:

```
public class Person {
   private String name;
   private int age;
   //... zde patří konstruktor
   public void writeInfo() {
      System.out.println("Person:");
      System.out.println("Name "+name);
      System.out.println("Age  "+age);
   }
}
```

## 6.6 Volání metod

- *Samotnou deklarací* (napsáním kódu) metody *se žádný kód neprovede*.
- Chceme-li vykonat kód metody, musíme ji *zavolat*.
- Volání se realizuje (tak jako u proměnných) "*tečkovou notací*", viz dále.
- Volání lze provést, jen je-li metoda z místa volání viditelná (přístupná). Přístupnost regulují podobně jako u proměnných modifikátory.

## 6.7 Volání metod – příklad

Vracíme se k prvnímu příkladu: vytvoříme dva lidi a zavoláme postupně jejich metodu *writeInfo*.

```
public class TestLidi {
   public static void main(String[] args) {
      Person ales = new Person("Ales Necas", 38);
      Person beata = new Person("Beata Novakova", 36);
      ales.writeInfo(); // volání metody objektu ales
      beata.writeInfo(); // volání metody objektu beata
   }
}
```

Vytvoří se dva objekty `Person` a vypíší se informace o nich.

## 6.8 Návrat z metody

Kód metody skončí, tj. předá řízení zpět volající metodě (nebo operačnímu systému v případě startovní metody `main`), jakmile

- dokončí poslední příkaz v těle metody nebo
- dospěje k příkazu `return`

Metoda může při návratu *vrátit hodnotu* - tj. chovat se jako *funkce* (ve pascalském smyslu):

- Vrácenou hodnotu musíme uvést za příkazem return. V tomto případě tedy nesmí return chybět!
- Typ vrácené hodnoty musíme *v hlavičce metody deklarovat.*
- Nevrací-li metoda nic, pak musíme namísto typu vracené hodnoty psát `void`.

Pozn.: I když metoda něco vrátí, my to nemusíme použít, ale je to trochu divné...

# 7 Metody – parametry

## 7.1 Parametry

- V deklaraci metody uvádíme v její hlavičce tzv. *formální parametry.* Syntaxe:
- ```
  modifikatory typVraceneHodnoty nazevMetody(seznamFormalnichParametru) {
     tělo (výkonný kód) metody
  }
  ```
- seznamFormalnichParametru je tvaru: typParametru nazevFormalnihoParametru, ...
- Podobně jako v jiných jazycích parametr představuje v rámci metody *lokální proměnnou.*
  ```

- Při volání metody jsou formální parametry nahrazeny *skutečnými parametry.*

## 7.2 Předávání skutečných parametrů metodám

Hodnoty *primitivních typů* - čísla, logické hodnoty, znaky

- se předávají **hodnotou**, tj. hodnota se nakopíruje do lokální proměnné metody

Hodnoty *objektových typů* - všechny ostatní (tj. vč. všech uživatelem definovaných typů)

- se předávají **odkazem**, tj. do lokální proměnné metody se nakopíruje **odkaz na objekt - skutečný parametr**Pozn: ve skutečnosti se tedy parametry *vždy předávají hodnotou*, protože v případě objektových parametrů se předává *hodnota odkazu na objekt - skutečný parametr.*

V Javě tedy nemáme jako programátoři moc na výběr, jak parametry předávat

- to je ale spíše výhoda!

## 7.3 Příklad předávání parametrů – primitivní typy

Rozšiřme definici třídy *Person* o metodu *scream* s parametry:

```
...
public void scream(int howManyTimes) {
   System.out.println("Kricim " + howManyTimes + "krat UAAAA!");
}
...
```

Při zavolání:

```
scream(10);
```

tato metoda vypíše

```
Kricim 10krat UAAAA!
```

## 7.4 Předávání parametrů – objektové typy (1)

Následující třída `Account` modeluje jednoduchý bankovní účet s možnostmi:

- přidávat na účet/odebírat z účtu
- vypisovat zůstatek na něm
- převádět na jiný účet

### 7.5 Předávání parametrů - objektové typy (2)

```
public class Account {
    // stav (zustatek) penez uctu
    private double balance;
    public void add(double amount) {
        balance += amount;
    }
    public void writeBalance() {
        System.out.println(balance);
    }
    public void transferTo(Account whereTo, double amount) {
        balance -= amount;
        whereTo.add(amount);
    }
}
```

Metoda *transferTo* pracovat nejen se svým "mateřským" objektem, ale i s objektem *whereTo* předaným do metody... opět přes tečkovou notaci.

### 7.6 Předávání parametrů – příklad 2

Příklad použití třídy *Account*:

```
...
public static void main(String[] args) {
   Account petrsAccount = new Account();
   Account ivansAccount = new Account();
   petrsAccount.add(100);
   ivansAccount.add(220);
   petrsAccount.transferTo(ivansAccount, 50);
   petrsAccount.writeBalance();
   ivansAccount.writeBalance();
}
```

## 8 Konstruktory

### 8.1 Konstruktory – co a k čemu?

- Konstruktury jsou speciální *metody* volané při *vytváření nových instancí* dané třídy.
- Typicky se v konstruktoru *naplní (inicializují) proměnné objektu.*
- Konstruktory lze volat jen ve spojení s operátorem `new` k vytvoření nové instance třídy - nového objektu, evt. volat z jiného konstruktoru

### 8.2 Konstruktory – syntaxe

```
public class Person {
   private String name;
```

```
   private int age;
   // konstruktor se dvěma parametry
   // - inicializuje hodnoty proměnných ve vytvořeném objektu
   public Person(String name, int age) {
      this.name = name;
      this.age = age;
   }
   ...
}
```

### 8.3 Konstruktory – použití

Příklad využití tohoto konstruktoru:

```
...
Person pepa = new Person("Pepa z Hongkongu", 105);
...
```

Toto volání vytvoří objekt pepa a naplní ho jménem a věkem.

### 8.4 Konstruktory – shrnutí

Jak je psát a co s nimi lze dělat?

- **nemají návratový typ** (ani `void` - *to už vůbec ne*!!!)
- mohou mít **parametry**
- mohou volat **konstruktor rodičovské třídy** pomocí klíčového slova `super`, které zastupuje jméno konstruktoru předka – to nás bude zajímat, jakmile začneme používat dědičnost
- Konstruktor předka se volá v **každém** případě, vždy jen jako svůj první příkaz a to, i když neuvedeme super() - pak se bezparametrický konstruktor zavolá automaticky.

## 9 Přetěžování metod

### 9.1 Přetěžování

- Jedna třída může mít více metod se *stejnými názvy, ale různými parametry.*
- Pak hovoříme o tzv. *přetížené* (overloaded) metodě.
- Nelze přetížit metodu *pouze změnou typu návratové hodnoty.*

### 9.2 Přetěžování – příklad

Ve třídě `Account` přetížíme metodu `transferTo`.

- Přetížená metoda převede na účet příjemce celý zůstatek z účtu odesílatele:

```
public void transferTo(Account whereTo) {
   whereTo.add(balance);
   balance = 0;
}
```

Ve třídě *Account* nyní koexistují dvě různé metody se stejným názvem, ale jinými parametry.Pozn: I když jsou to teoreticky dvě úplně různé metody, pak *když už se jmenují stejně, měly by dělat něco podobného.*

### 9.3 Přetěžování – příklad (2)

- Často přetížená metoda volá jiné "vydání" metody se stejným názvem:

  ```
  public void transferTo(Account whereTo) {
     transferTo(whereTo, balance);
  }
  ```

- Toto je *jednodušší, přehlednější*, udělá se tam potenciálně méně chyb.Lze doporučit. Je to přesně postup divide-et-impera, rozděl a panuj, dělba práce mezi metodami!

### 9.4 Přetěžování – příklad (3)

- Je ale otázka, zdali převod celého zůstatku raději nenapsat jako *nepřetíženou*, samostatnou metodu, např.:

  ```
  public void transferAllMoneyTo(Account whereTo) {
     transferTo(whereTo, balance);
  }
  ```

- Je to o něco instruktivnější, ale přibude další identifikátor - název metody - k zapamatování.Což může být výhoda (je to výstižné) i nevýhoda (musíme si pamatovat další).

## 10 Odkazy na objekty

### 10.1 Odkazy na objekty (instance)

Deklarace proměnné objektového typu sama o sobě *ještě žádný objekt nevytváří.*To se děje až příkazem (operátorem) new.

- Proměnné objektového typu jsou vlastně pouze **odkazy***na dynamicky vytvářené objekty.*
- Přiřazením takové proměnné zkopírujeme pouze odkaz. Na jeden objekt se odkazujeme nadále ze dvou míst. Nezduplikujeme tím celý objekt!

## 10.2 Přiřazování objektových proměnných (1)

V následující ukázce vytvoříme dva účty.

- Odkazy na ně budou primárně v proměnných *petrsAccount* a *ivansAccount*.
- V proměnné u nebude primárně odkaz na žádný účet.
- Pak do ní přiřadíme (p = petrsAccount;) odkaz na objekt skrývající se pod odkazem *petrsAccount*.
- Od této chvíle můžeme s účtem *petrsAccount* manipulovat rovněž přes odkaz (proměnnou) *p*.Což se také děje: p.transferTo(ivansAccount, 50);

## 10.3 Přiřazování objektových proměnných (2)

```
 ...
   public static void main(String[] args) {
      Account petrsAccount = new Account();
      Account ivansAccount = new Account();
      Account p;
      petrsAccount.add(100);
      ivansAccount.add(220);
      p = petrsAccount;
      p.transferTo(ivansAccount, 50); //  odečte se z také Petrova účtu
      petrsAccount.writeBalance(); // vypíše 50
      ivansAccount.writeBalance();
 }
```

## 10.4 Vracení odkazu na sebe (1)

Metoda může vracet odkaz na objekt, nad nímž je volána pomocí `return this;`
Příklad - upravený `Account` s metodou `transferTo` vracející odkaz na sebe

## 10.5 Vracení odkazu na sebe (2)

```
public class Account {
   private double balance;
   public void add(double amt) {
      balance += amt;
   }
   public void writeBalance() {
      System.out.println(balance);
   }
   public Account transferTo(Account whereTo, double a) {
      add(-a);
      whereTo.add(a);
      return this;
   }
}
```

## 10.6 Řetězení volání

Vracení odkazu na sebe (tj. na objekt, na němž se metoda volala) lze s výhodou využít k "řetězení" volání:

```
 ...
   public static void main(String[] args) {
      Account petrsAccount = new Account();
      Account ivansAccount = new Accont();
      Account igorsAccount = new Account();
      petrsAccount.add(100);
      ivansAccount.add(100);
      igorsAccount.add(100);
// budeme řetězit volání:
      petrsAccount.transferTo(ivansAccount, 50).transferTo(igorsAccount, 20);
      petrsAccount.writeBalance(); // vypíše 30
      ivansAccount.writeBalance(); // vypíše 150
      igorsAccount.writeBalance(); // vypíše 120
   }
```