Seminář Java
II
2005/2006

Radek Kočí

Rekapitulace

- `http://www.fit.vutbr.cz/study/courses/IJA/`
- Diskuzní fórum `vutbr.fit.courses.ija`
- Java – úvod, historie, distribuce
- Základy objektové orientace

Téma přednášky

- Vytvoření a spuštění aplikace – demonstrační příklad
- Balíky – organizace tříd
- Třídy a rozhraní – deklarace, použití
- Konstruktory
- Datové typy

Základní životní cyklus programu v Javě

- Program sestává z alespoň jedné třídy
- Zdrojový kód každé veřejně přístupné třídy je umístěn v jednom souboru
  - `NazevTridy.java`
  - přípona `.java` je povinná!
- Postup:
  - vytvoření zdrojového textu (libovolný editor) $\Rightarrow$ `Pokus.java`
  - překlad: `Pokus.java` $\Rightarrow$ `Pokus.class`
  - spuštění: `java Pokus`

- Překlad
  - `javac název_souboru_s_třídou`
  - název souboru se udává včetně přípony `.java`
- Spuštění
  - `java název_třídy`
  - bez přípony `.class`

Ukázka aplikace

Ještě než skutečně začneme

- veřejná třída odpovídá jednomu souboru
- třídy jsou členěny do balíků (package)
- u běžné "desktopové" aplikace představuje vstupní bod do programu třída obsahující metodu `main`
- Java je *case sensitive*! (`ucet` x `Ucet`)
- nezbytná pomůcka při programování v Javě: `http://java.sun.com/reference/api/index.html`

Soubor `Pozdrav.java`:

```
package IJA.seminar1;
public class Pozdrav {
    // Program spouštíme aktivací funkce "main"
    public static void main(String[] args) {
        System.out.println("Ahoj!");
    }
}
```

Co znamená spustit program?

Spuštění javového programu odpovídá spuštění metody *main* jedné ze tříd tvořících program

Aplikace může mít parametry:

- podobně jako např. v Pascalu nebo v C
- jsou typu `String` (řetězec)
- předávají se při spuštění z příkazového řádku do pole `String[] args` (argument metody *main*)

Metoda `public static void main(String[] args)`

- nevrací žádnou hodnotu – návratový typ je vždy(!) void
- její hlavička musí vypadat vždy přesně tak, jako ve výše uvedeném příkladu, jinak nebude spuštěna!

Organizace tříd do balíků

- třídy jsou členěny do balíků (package)
- organizaci balíků odpovídá organizace adresářů a umístění zdrojového souboru do příslušného adresáře

---

```
package IJA.seminar1;
```

```
$HOME
  |-- examples
       |-- IJA
            |-- seminar1
                 |-- Pozdrav.java
```

Překlad

1. jsme v adresáři `$HOME/examples`
2. spustíme překlad `javac IJA/seminar1/Pozdrav.java`
3. je-li program správně napsán, přeloží se "mlčky"
4. přeložený soubor `Pozdrav.class` bude v témže adresáři jako zdroj

Organizace tříd do balíků

Volba `classpath` pro `javac` a `java`, systémová proměnná `CLASSPATH`

- definují adresáře tvořící "kořenový" adresář pro hledání balíků a tříd
- třídy (soubory .class) se budou hledat v odpovídajících podadresářích uvedeného adresáře (adresářů)

```
$HOME
  |-- java
       |-- distribution
       |-- project
       |-- docs
  |-- sun
       |-- distribution
       |-- examples
       |-- docs
```

Kořenový adresář:
`$HOME/java/project`
`$HOME/sun/examples`

```
export CLASSPATH="$CLASSPATH:$HOME/java/project:..."

javac -classpath "$HOME/java/project:..." ...
java -classpath "$HOME/java/project:..." ...
```

Ukázka aplikace

Soubor `Pozdrav.java`:

```
package IJA.seminar1;
public class Pozdrav {
    // Program spouštíme aktivací funkce "main"
    public static void main(String[] args) {
        System.out.println("Ahoj!");
    }
}
```

```
$HOME
  |-- examples
       |-- IJA
            |-- seminar1
                 |-- Pozdrav.java
```

Překlad: `javac -classpath "$HOME/examples" $HOME/examples/IJA/seminar1/Pozdrav.java`

Spuštění: `java -classpath "$HOME/examples" IJA.seminar1.Pozdrav`

Rozhraní, třída a objekt v Javě

Rozhraní

- definuje *typ* objektu
- specifikuje množinu vlastností, ale *neimplementuje je*
- vlastnostmi se myslí především metody
- rozhraní v Javě tvoří
  - množina hlaviček metod označená identifikátorem – názvem rozhraní
  - + ucelené specifikace – tj. popis, co přesně má metoda dělat (vstupy/výstupy metody, její vedlejší efekty . . .)

Rozhraní, třída a objekt v Javě

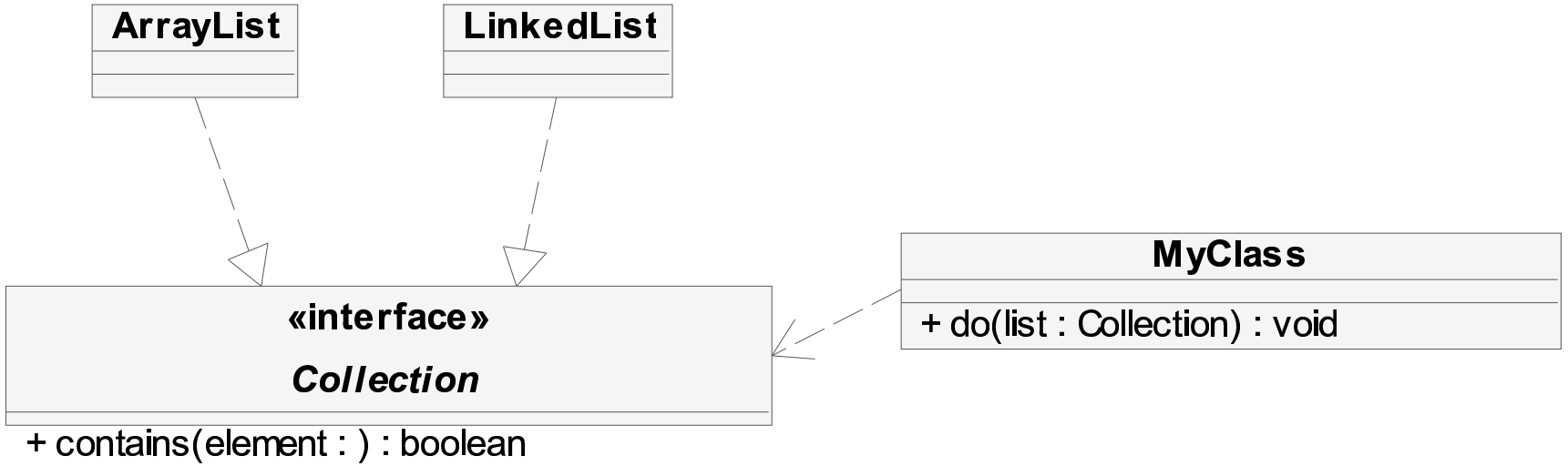
Třída (také poněkud nepřesně zvaná objektový typ)

- implementuje rozhraní (tj. všechny metody rozhraní)
- třída představuje vzor pro vytváření objektů
- třída představuje skupinu objektů, které nesou stejné vlastnosti (kvalitativně)
  - např. všechny objekty třídy `Clovek` mají vlastnost `jmeno`,
  - tato vlastnost má však obecně pro různé lidi různé hodnoty – lidi mají různá jména
- *pozn.: třída sama o sobě deklaruje rozhraní $\Rightarrow$ třída také definuje typ objektu*

Objekt

- objekt je jeden konkrétní jedinec (reprezentant, entita) příslušné třídy
- pro konkrétní objekt nabývají vlastnosti deklarované třídou konkrétních hodnot
- objekt je *instancí* třídy

Rozhraní, třída a objekt v Javě




```
public void do(Collection list) {
    if (list.contains(...)) ...
}

obj = new MyClass();
obj.do(new LinkedList());
obj.do(new ArrayList());
```

Vlastnosti objektu

Vlastnosti objektů je třeba deklarovat

- proměnné
  - jsou nositeli "pasivních" vlastností, charakteristik objektů
  - datové hodnoty svázané (zapouzdřené) v objektu
- metody
  - jsou nositeli "výkonných" vlastností, "dovedností" objektů
  - v podstatě funkce (procedury) pracující primárně s proměnnými "mateřského" objektu
  - může mít další parametry (argumenty metody)
  - může vracet hodnotu
  - v Javě neexistují metody deklarované mimo třídy

Deklarace třídy

```
modifikátory class názevTřídy [extends, implements]
{
      tělo třídy
      // deklarace proměnných objektu
      // deklarace metod
}
```

Např.:

```
public class Ucet
{
}
```

Modifikátory

- `public`
- `private`
- `protected`
- *žádný*

Deklarace proměnné objektu

Deklarace proměnné objektu má tvar:

```
modifikátory Typ jméno;
```

např.:

```
protected double castka;
```

Jmenné konvence

- jména začínají malým písmenem
- nepoužíváme diakritiku (problémy s editory, přenositelností a kódováním znaků)
- (raději ani český jazyk, angličtině rozumí "každý")
- je-li to složenina více slov, pak je nespojujeme podtržítkem, ale další začne velkým písmenem

Deklarace metody

```
modifikátory typVrácenéHodnoty
                          nazevMetody ( seznamFormPar )
{
      tělo (výkonný kód) metody
}

seznamFormParam = typ názevFormParametru, ...
```

Např.:

```
public void prevedNa(Ucet kam, double castka) {
    uber(castka);
    kam.pridej(castka);
}
```

Deklarace proměnné v metodě:

```
Typ jméno;
```

Modifikátory přístupu

Přístup ke třídám i jejim prvkům lze (podobně jako např. v C++) regulovat:

- přístupem se rozumí jakékoli použití dané třídy, prvku, . . .
- omezení přístupu je kontrolováno hned při překladu
- takto lze regulovat přístup staticky, mezi celými třídami, nikoli pro jednotlivé objekty

Granularita omezení přístupu

- přístup je v Javě regulován jednotlivě po prvcích
- omezení přístupu je určeno uvedením jednoho z modifikátoru přístupu (access modifier) nebo neuvedením žádného

Modifikátory přístupu

Typy omezení přístupu

- `public` = veřejný
- `protected` = chráněný
- `private` = soukromý
- modifikátor neuveden = říká se lokální v balíku nebo chráněný v balíku nebo "přátelský"

Pro třídy:

- veřejné (*public*)
  - přístup k třídě není omezen
- neveřejné (*lokální v balíku*)
  - k třídě může přistupovat libovolná třída ze stejného balíku

Modifikátory přístupu

Pro vlastnosti tříd (proměnné/metody):

- veřejné (*public*)
- chráněné (*protected*)
  - přístupné jen ze tříd stejného balíku a z podtříd (i když jsou v jiném balíku)
- neveřejné (*lokální v balíku*)
  - přístupné jen ze tříd stejného balíku, už ale ne z podtříd umístěných v jiném balíku (nedoporučuje se)
- soukromé (*private*)
  - přístupné jen v rámci třídy – používá se častěji pro proměnné než metody
  - zneviditelníme i případným podtřídám

Ukázka deklarace třídy

```
package IJA.seminar2.banka.ucty;

public class Ucet {
    protected String majitel;
    protected double zustatek;

    public void pridej(double castka) {
        zustatek += castka;
    }
    public void vypisZustatek() {
        System.out.println(zustatek);
    }
    public void uber(double castka) {
        zustatek -= castka;
    }
    public void prevedNa(Ucet kam, double castka) {
        uber(castka);
        kam.pridej(castka);
    }
}
```

Deklarace rozhraní

- Vypadá i umisťuje se do souborů podobně jako deklarace třídy
- Všechny metody v rozhraní musí být public a v hlavičce se to ani nemusí uvádět.
- Všechny metody v rozhraní jsou zároveň automaticky abstraktní $\Rightarrow$ těla metod se neuvádějí.
- Rozhraní může obsahovat proměnné – jedná se vždy o konstantu (modifikátor `final` se uvádět nemusí)

Příklad deklarace rozhraní

```
public interface Informator {
     public void vypisInfo();
}
```

Implementace rozhraní

```
public class Ucet implements Informator {
     ...
     public void vypisInfo() {
          ...
     }
}
```

- Třída implementuje všechny metody předepsané rozhraním.
- Třída může implementovat více rozhraní současně.

```
public class Name implements Interface1, Interface2

{ ...  }
```

Datové typy

Java striktně rozlišuje mezi hodnotami

- primitivních datových typů
  - čísla
  - logické hodnoty
  - znaky
- objektových typů
  - řetězce
  - uživatelem definované typy – třídy a rozhraní

Základní rozdíl je v práci s proměnnými:

- proměnné primitivních datových typů přímo obsahují danou hodnotu
- proměnné objektových typů obsahují pouze odkaz na příslušný objekt
- $\Rightarrow$ dvě objektové proměnné mohou nést odkaz na tentýž objekt

Primitivní datové typy

- Proměnné těchto typů nesou atomické, dále nestrukturované hodnoty
- Deklarace způsobí
  - rezervování příslušného paměťového prostoru
  - zpřístupnění (pojmenování) tohoto prostoru identifikátorem proměnné
- inicializace $\Rightarrow$ `0`
  `int pocetUctu;` $\Leftarrow pocetUctu == 0$

Typ `boolean`

- logická hodnota, přípustné hodnoty jsou `false` a `true`
- na rozdíl od Pascalu na nich není definováno uspořádání

Typ `void`

- není v pravém slova smyslu datovým typem, nemá žádné hodnoty
- označuje "prázdný" typ pro sdělení, že určitá metoda nevrací žádný výsledek

Primitivní datové typy

Čísla s pohyblivou řádovou čárkou

- `float`
  - 32 bitů
- `double`
  - 64 bitů
- zápis literálů
  - `float f = -.777f, g = 0.123f, h = -4e6f, i = 1.2E-15f;`
  - `double f = -.777, g = 0.123, h = -4e6, i = 1.2E-15;`

Primitivní datové typy

Integrální typy – celočíselné

- v Javě jsou celá čísla vždy interpretována jako znaménková
- `int`
  - 32 bitů ($-2\,147\,483\,648$ .. $2\,147\,483\,647$)
  - základní celočíselný typ
- `long`
  - 64 bitů (cca $+/- \ 9 \cdot 10^{18}$)
- `short`
  - 16 bitů (-32768 .. 32767)
- `byte`
  - 8 bitů (-128 .. 127)

Primitivní datové typy

Integrální typy – `char`

- char představuje jeden 16bitový znak v kódování UNICODE
- konstanty typu char zapisujeme
  - v apostrofech: `'a'`, `'ř'`
  - pomocí escape-sekvencí: $\backslash n$ (konec řádku) $\backslash t$ (tabulátor)
  - hexadecimálně: $\backslash u0040$ (totéž, co `'a'`)
  - oktalově: $\backslash 127$
- *Pozor na kódové stránky při překladu/spouštění – dochází k překódování textu! (komentář, znak, řetězec, identifikátor)*
  `javac -encoding ISO8859-2 ...`

Proměnné objektového typu

- Proměnné těchto typů reprezentují reference na objekty
- Deklarace způsobí
  - rezervování paměťového prostoru na *referenci*!
  - vlastní objekt (instance třídy) nevzniká!
- inicializace $\Rightarrow$ `null`
  `Ucet ucet;` $\Leftarrow ucet == null$

```
public class Banka {
    public static void main(String[] args) {
        Ucet ucet;
        Ucet jinyUcet;
        Ucet uplneJinyUcet = ucet;
    }
}
```

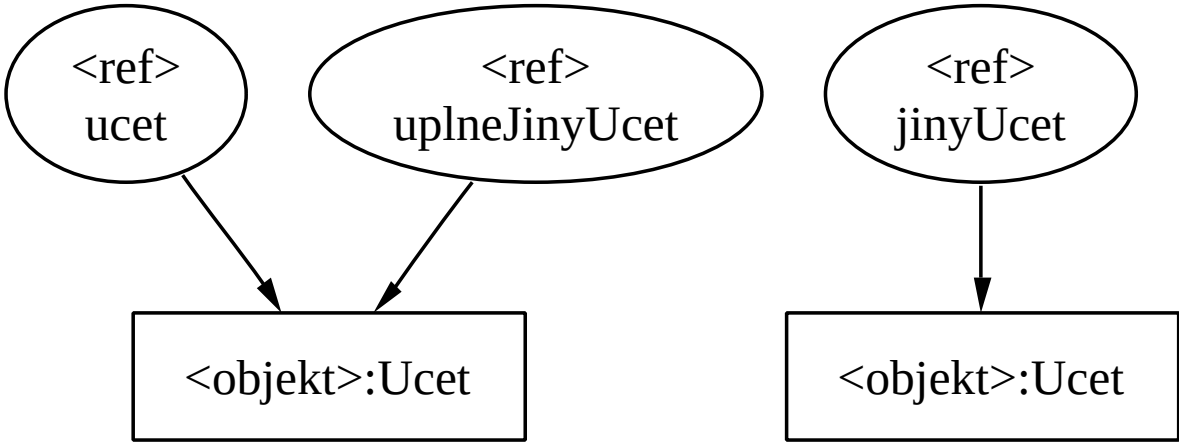
Získali jsme 3 prázdné (null) *reference*.

Proměnné objektového typu

Vytvoření instance

- operátor new
- rezervuje paměťový prostor pro objekt (instanci dané třídy)

```
public class Banka {
    public static void main(String[] args) {
        Ucet ucet = new Ucet();
        Ucet jinyUcet = new Ucet();
        Ucet uplneJinyUcet = ucet;
    }
}
```



Použití třídy

deklarace (určení typu) proměnné

- `Typ jmeno;`
- `Ucet ucet;`
- `ucet` bude typu `Ucet`

vytvoření objektu

- `ucet = new Ucet();`
- vytvoří se objekt třídy `Ucet` a uloží se (resp. *reference na objekt*) do proměnné `ucet`

sloučení deklarace a inicializace na jeden řádek

- `Ucet ucet = new Ucet();`

Volání metod

Nad existujícími (vytvořenými) objekty můžeme volat jejich metody

- 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"
- volání lze provést, jen je-li metoda z místa volání přístupná
- přístupnost regulují podobně jako u proměnných modifikátory přístupu

```
public class Banka {
    public static void main(String[] args) {
        Ucet ucet = new Ucet();
        ucet.vypisZustatek();
        ucet.pridej(100.50);
        ucet.vypisZustatek();
        ucet.uber(0.50);
        ucet.vypisZustatek();
    }
}
```

Předávání parametrů metodám

Hodnoty primitivních typů

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

Hodnoty objektových 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 se buď předává kopie hodnoty primitivního typu, nebo kopie hodnoty odkazu (reference) na objekt.

Návrat z metody

Kód metody skončí jakmile

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

Metoda může při návratu vrátit hodnotu (chovat se jako funkce)

- vrácenou hodnotu musíme uvést za příkazem `return`
- typ vrácené hodnoty musíme v hlavičce metody deklarovat
- nevrací-li metoda nic, pak musíme namísto typu vracené hodnoty psát `void` (v tomto případě se `return` nemusí uvádět)

Ukončení metody způsobí předání řízení

- zpět volající metodě + předání případné hodnoty
- systému (JVM) v případě ukončení metody `main`

Přetěžování metod

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.

```
public void prevedNa(Ucet kam, double castka) {
      uber(castka);
      kam.pridej(castka);
}


public void prevedNa(Ucet kam) {
      prevedNa(kam, zustatek);
}
```

Vracení odkazu na sebe

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

```
public Ucet prevedNa(Ucet kam, double castka) {
      uber(castka);
      kam.pridej(castka);
      return this;
}
```

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) {
     Ucet petruvUcet = new Ucet();
     Ucet ivanuvUcet = new Ucet();
     petruvUcet.pridej(100);
     ivanuvUcet.pridej(100);

     // budeme řetězit volání:
     petruvUcet.prevedNa(ivanuvUcet, 30).vypisZustatek();
     // převede 30 jednotek a vypíše zůstatek => 70

     ivanuvUcet.vypisZustatek(); // vypíše 130
}
```

Vytváření objektů

Voláním např. `new Ucet()` jsme použili:

- operátor `new`, který vytvoří prázdný objekt a
- volání *konstruktoru*, který prázdný objekt naplní počátečními údaji (daty).

Konstruktory

- 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, eventuálně volat z jiného konstruktoru.

Implicitní konstruktor

Každá třídá má *implicitní* (*bezparametrický*) konstruktor

- nemá žádné parametry
- nemá žádný návratový typ!
- nemusí se deklarovat
- deklarace: `JmenoTridy() {...}`

```
public class Ucet {
    public Ucet() {
        ...
    }
}
```

Použití: `new Ucet();`

Další konstruktory

Každá třída může mít další (jiné) konstruktory než implicitní

- odlišují se parametry
- nemají návratový typ
- pokud se deklaruje alespoň jeden konstruktor, implicitní se již negeneruje!!

```
public class Ucet {
    private String majitel;
    public Ucet(String name) {
        majitel = name;
    }
}
```

Použití:

```
new Ucet("Ferda");
new Ucet();
```

$\Leftarrow$ chyba!

Další konstruktory

Pokud chceme deklarovat další konstruktory a současně používat implicitní, musíme ho také deklarovat!

```
public class Ucet {
private String majitel;
    public Ucet() {}
    public Ucet(String name) {
        majitel = name;
    }
}
```

Použití:

```
new Ucet("Ferda");
new Ucet();
```

⇐ OK

Proměnné a metody třídy – statické

Dosud jsme zmiňovali proměnné a metody (tj. souhrnně prvky – members) objektu.

- Lze deklarovat také metody a proměnné patřící celé třídě, tj. skupině všech objektů daného typu.
- Takové metody a proměnné nazýváme statické a označujeme v deklaraci modifikátorem `static`

Příklad – počítání účtů

```
public class Ucet {
    protected String majitel;
    protected double zustatek = 0;
    protected static int pocet = 0;

    public Ucet(String jmeno) {
        majitel = jmeno;
        pocet++;
    }

    public static int pocetUctu() {
        return pocet;
    }
}
```

```
Ucet ucet = new Ucet("Ferda");
System.println(Ucet.pocetUctu());
```

Příklad – počítání účtů (volání konstruktoru)

```
public class Ucet {
    protected String majitel;
    protected double zustatek = 0;
    protected static int pocet = 0;

    public Ucet() {
        pocet++;
    }
    public Ucet(String jmeno) {
        this();
        majitel = jmeno;
    }

    public static int pocetUctu() {
        return pocet;
    }
}
```

Shrnutí

Objekty:

- jsou instance "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)

Odkazy na objekty

- Deklarace proměnné objektového typu ještě žádný objekt nevytváří.
- To se děje až příkazem (operátorem) – `new`.
- Proměnné objektového typu jsou vlastně odkazy na dynamicky vytvořené objekty.
- Přiřazením takové proměnné zkopírujeme pouze odkaz, nikoli celý objekt.

Komentáře

- Základní typy komentářů (podobně jako např. v C/C++)
  - *řádkové* od značky `//` do konce řádku
  - *blokové* (na libovolném počtu řádků) začínají `/*` pak je text komentáře, končí `*/`
  - *dokumentační* (na libovolném počti řádků) od značky `/**` po značku `*/` Každý další řádek může začínat mezerami či *, hvězdička se v komentáři neprojeví.

```
// řádkový komentář
/*
   blokový
   (víceřádkový) komentář
*/
/**
   dokumentační
   (víceřádkový) komentář
*/
```

Ukázka použití dokumentačních komentářů

```
package IJA.seminar2.banka.ucty;

/**
 * Trida ucet
 * @author R. Koci
**/
public class Ucet {
    /** Majitel uctu */
    protected String majitel;
    protected double zustatek;

    public void pridej(double castka) {
        zustatek += castka;
    }
    public void vypisZustatek() {
        System.out.println(zustatek);
    }
    public void uber(double castka) {
        zustatek -= castka;
    }
}
```

Generování dokumentace

```
javadoc -classpath "..." IJA.seminar2.banka.ucty
```

Dokumentace

- je generována nástrojem `javadoc`
  1. z dokumentačních komentářů
  2. a ze samotného zdrojového textu
- *je tedy možné dokumentovat (základním způsobem) i program bez vložených komentářů!*
- má standardně podobu HTML stránek (s rámy i bez)
- chování `javadoc` můžeme změnit volbami (options) při spuštění

Dokumentační komentáře uvádíme:

- před hlavičkou třídy (komentuje třídu jako celek)
- před hlavičkou metody nebo proměnné (komentuje příslušnou metodu nebo proměnnou)

Značky pro dokumentační komentáře

`javadoc` můžeme podrobněji instruovat pomocí značek vkládaných do dokumentačních komentářů, např.:

@author specifikuje autora API/programu

@version označuje verzi API, např. "1.0"

@deprecated informuje, že prvek je zavrhovaný

@exception popisuje informace o výjimce, kterou metoda propouští ("vyhazuje")

@param popisuje jeden parametr metody

@since uvedeme, od kdy (od které verze pg.) je věc podporována/přítomna

@see uvedeme odkaz, kam je také doporučeno nahlédnout (související věci)