Seminář Java

III

Rekapitulace

- Deklarace tříd
- Proměnné, metody, modifikátory přístupu
- Konstruktory
- Datové typy
- Balíky

Dědičnost

Co už víme ...

- Třídy popisují skupiny objektů podobných vlastností
- Třídy mohou mít tyto skupiny vlastností:
  - Metody – procedury/funkce, které pracují (především) s objekty této třídy
  - Proměnné – pojmenované datové prvky (hodnoty) uchovávané v každém objektu této třídy
- Vlastnosti jsou ve třídě "schované" (zapouzdřené)

Dědičnost

Dědičnost

- Specializace, rozšiřování funkčnosti třídy.
- Odvození nové třídy od nějaké stávající
- Odvozená (dceřinná) třída
  - má všechny vlastnosti nadtřídy
  - + vlastnosti uvedené přímo v deklaraci podtřídy
  - *Konstruktory se nedědí!!!*

Třída Ucet

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

    public Ucet(String name);
    public void pridej(double castka);
    public void vypisZustatek();
    public void uber(double castka);
    public void prevedNa(Ucet kam, double castka);
    public void prevedNa(Ucet kam);
}
```

Třída KUcet

```
public class KUcet extends Ucet {
    protected double kkorent;

    public KUcet(String name, double kk) {
        ...
    }

    public boolean uber(double castka) {
        ...
    }
}
```

Inicializace objektu

Základní kroky

1. nalezení a vyvolání konstruktoru
2. vyvolání bezparametrického konstruktoru nadřazené třídy
3. inicializace instančních proměnných
4. provedení těla konstruktoru třídy

Inicializace objektu – příklad

```
B b = new B();
```

```
class Z {
    public Z() {System.out.println("Konstr.  Z");}
}

class A {
    public A() {System.out.println("Konstr.  A");}
}

class B extends A {
    Z z = new Z();
    public B() {System.out.println("Konstr.  B");}
}
```

Inicializace objektu – příklad

```
B b = new B();
```

```
class Z {
    public Z() {System.out.println("Konstr.  Z");}
}

class A {
    public A() {System.out.println("Konstr.  A");}
}

class B extends A {
    Z z = new Z();
    public B() {System.out.println("Konstr.  B");}
}
```

Inicializace objektu – příklad

```
B b = new B();
```

```
class Z {
    public Z() {System.out.println("Konstr.  Z");}
}

class A {
    public A() {System.out.println("Konstr.  A");}
}

class B extends A {
    Z z = new Z();
    public B() {System.out.println("Konstr.  B");}
}
```

Inicializace objektu – příklad

```
B b = new B();
```

```
class Z {
    public Z() {System.out.println("Konstr.  Z");}
}

class A {
    public A() {System.out.println("Konstr.  A");}
}

class B extends A {
    Z z = new Z();
    public B() {System.out.println("Konstr.  B");}
}
```

```
Konstr.  A
```

Inicializace objektu – příklad

```
B b = new B();
```

```
class Z {
    public Z() {System.out.println("Konstr.  Z");}
}

class A {
    public A() {System.out.println("Konstr.  A");}
}

class B extends A {
    Z z = new Z();
    public B() {System.out.println("Konstr.  B");}
}
```

```
Konstr.  A
```

Inicializace objektu – příklad

```
B b = new B();
```

```
class Z {
    public Z() {System.out.println("Konstr.  Z");}
}

class A {
    public A() {System.out.println("Konstr.  A");}
}

class B extends A {
    Z z = new Z();
    public B() {System.out.println("Konstr.  B");}
}
```

```
Konstr.  A
Konstr.  Z
```

Inicializace objektu – příklad

```
B b = new B();
```

```
class Z {
    public Z() {System.out.println("Konstr.  Z");}
}

class A {
    public A() {System.out.println("Konstr.  A");}
}

class B extends A {
    Z z = new Z();
    public B() {System.out.println("Konstr.  B");}
}
```

```
Konstr.  A
Konstr.  Z
Konstr.  B
```

Inicializace objektu – II

Možné modifikace

- lze volat jiný než bezparametrický konstruktor nadřazené třídy (musí být vždy na začátku konstruktoru potomka), např.
  `super(parametry)`
- lze volat i jiný konstruktor třídy (musí být vždy na začátku konstruktoru), např.
  `this(parametry)`
- *bezparametrický (implicitní) konstruktor neexistuje, pokud existuje alespoň jeden jiný*

`super` a `this` lze použít i pro volání metod nadřazené/dané třídy

Třída KUcet

```
public class KUcet extends Ucet {
    protected double kkorent;

    public KUcet(String name, double kk) {
        super(name);
        kkorent = kk;
    }

    public boolean uber(double castka) {
        if ((zustatek+kkorent) >= castka) {
            super.uber(castka);
            return true;
        }
        else
            return false;
    }
}
```

Typy omezení přístupu

Pro vlastnosti tříd = proměnné/metody:

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

Rekapitulace

- Umíme deklarovat třídu a její vlastnosti.
- Umíme vytvářet instance tříd a volat její metody.
- Umíme vytvářet specializované třídy a rozhraní.
- Umíme implementovat rozhraní.

Příkazy v Javě

Co už známe ...

- volání metody
- návrat z metody (`return`)
- příkaz je ukončen středníkem (`;`)

Nové ...

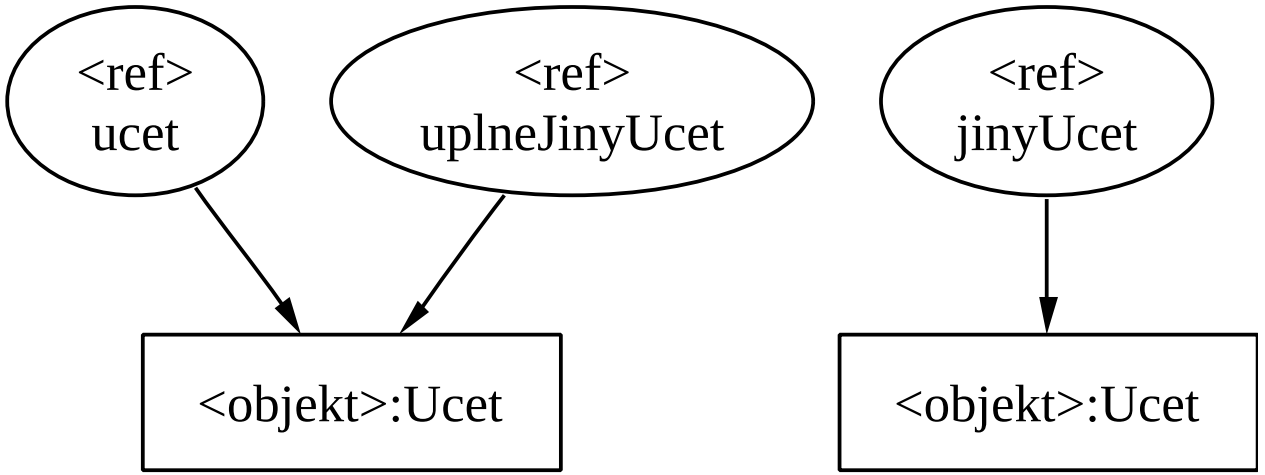
- přiřazovací příkaz (`=`)
- řízení toku programu

Přiřazení

- Na levé straně musí být proměnná.
- Na pravé straně musí být *přiřaditelný* výraz.
- Primitivní typy
  - přiřazením se hodnota zduplikuje
  - konverze typů (`short` → `int`, `int` → `short`)
- Přiřazení odkazu na objekt
  - Proměnné objektového typu obsahují odkazy (reference) na objekty, ne objekty samotné!!!
  - přiřazením se duplikuje pouze reference

Přiřazení proměnné objektového typu

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



Pole

- Pole v Javě je speciálním objektem.
- Můžeme mít pole jak primitivních, tak objektových hodnot
  - pole primitivních hodnot tyto hodnoty obsahuje
  - pole objektů obsahuje odkazy na objekty
- Kromě pole v Javě existují i jiné objekty na ukládání více hodnot – *kontejnery* (bude později ...)

Pole – I

Před použitím je nutné pole

- deklarovat
- vytvořit
- inicializovat (naplnit)

Syntaxe deklarace

- `typhodnoty [] identifikator`
- na rozdíl od C/C++ nikdy neuvádíme při deklaraci počet prvků pole – ten je podstatný až při vytvoření objektu pole

Pole – II

Vytvoření pole

- jako u jiného objektu – voláním konstruktoru:
  - `nazevPole = new typhodnoty [počet_prvků];`
- nebo inicializací při deklaraci:
  - `int [] nazevPole = { 1, 2, 3};`

Syntaxe přístupu k prvkům

- `identifikator [ indexprvku ]`
- přiřazení prvku do pole:
  `identifikator [ indexprvku ] = hodnota;`
- čtení hodnoty z pole:
  `proměnná = identifikator [ indexprvku ];`

Pole – III

```
Ucet [] ucty;                       // deklarace pole
ucty = new Ucet[5];                 // vytvoření pole
ucty[0] = new Ucet("Franta");  // vytvoření objektu
                  // a inicializace 1. prvku pole !!!

ucty[0].vypisInfo();              // přístup k prvku pole
```

- V poli `ucty` je naplněn 1. prvek odkazem na objekt
- Ostatní prvky zůstaly naplněny prázdnými odkazy `null`.

Pole – IV

Co když vynecháme vytvoření pole?

```
Ucet [] ucty;
ucty[0] = new Ucet("Franta");  // chyba, pole neexistuje
```

Co když vynecháme inicializaci pole?

```
Ucet [] ucty;
ucty = new Ucet[5];
ucty[0].vypisInfo();          // chyba, prvek neexistuje
```

Kopírování pole

Přiřazení proměnné objektového typu (a tedy i polí) vede pouze k duplikaci odkazu, nikoli celého odkazovaného objektu.

```
Ucet [] ucty = new Ucet[5];
Ucet [] ucty2;
ucty2 = ucty;
```

- Proměnná `ucty2` obsahuje odkaz na stejné pole jako `ucty`.

```
Ucet [] ucty2 = new Ucet[5];
System.arraycopy(ucty, 0, ucty2, 0, lidi.length);
```

- Proměnná `ucty2` obsahuje kopii původního pole.
- Také `arraycopy` však do cílového pole zduplikuje jen odkazy na objekty, nevytvoří kopie objektů!

Řídící příkazy

- if
- while
- do – while
- for
- switch
- break, continue

Příkazy

Příkazy mohou být jednoduché

- `pole[i] = 20;`

nebo složené

- `{ pole[i] = 20; i++; }`

Řízení toku programu v těle metody

Příkaz (neúplného) větvení if

```
if (logický výraz) příkaz
```

- platí-li logický výraz (má hodnotu true), provede se příkaz

Příkaz úplného větvení if - else

```
if (logický výraz)
    příkaz1
else
    příkaz2
```

- platí-li logický výraz (má hodnoty `true`), provede se `příkaz1`
- neplatí-li, provede se `příkaz2`
- větev else se nemusí uvádět
- větvení `if - else` můžeme vnořovat do sebe

Cyklus s podmínkou na začátku

- Tělo cyklu se provádí tak dlouho, dokud platí podmínka

v těle cyklu je jeden jednoduchý příkaz ...

```
while (podmínka)
    příkaz;
```

... nebo příkaz složený

```
while (podmínka) {
    příkaz1;
    příkaz2;
    příkaz3;
    ...
}
```

- Tělo cyklu se nemusí provést ani jednou - pokud už hned na začátku podmínka neplatí

Doporučení k psaní cyklů/větvení

- Větvení, cykly: vždy psát se složeným příkazem v těle (tj. se složenými závorkami)!!!
- jinak hrozí, že se v těle větvení/cyklu z neopatrnosti při editaci objeví něco jiného, než chceme, např.:

```
while (i < a.length)
    System.out.println(a[i]); i++;
```

Pišme proto vždy takto:

```
while (i < a.length) {
    System.out.println(a[i]); i++;
}
```

Cyklus s podmínkou na konci

- Tělo se provádí dokud platí podmínka (vždy aspoň jednou).
- Obdoba repeat v Pascalu (podmínka je ovšem interpretována opačně).
- Relativně málo používaný - je méně přehledný než while

```
do {
     příkaz1;
     příkaz2;
     příkaz3;
     ...
} while (podmínka);
```

Cyklus "for"

- obecnější než for v Pascalu, podobně jako v C/C++
- de-facto jde o rozšíření while, lze jím snadno nahradit

```
for (počáteční operace; vstupní podmínka;
                        příkaz po každém průchodu)
    příkaz;
```

anebo (obvyklejší, bezpečnější)

```
for (počáteční operace; vstupní podmínka;
                        příkaz po každém průchodu)
{
    příkaz1;
    příkaz2;
    příkaz3;
    ...
}
```

Příklad použití "for" cyklu

Provedení určité sekvence určitý počet krát

```
for (int i = 0; i < 10; i++) {
    System.out.println(i);
}
```

- Vypíše na obrazovku deset řádků s čísly postupně 0 až 9

Ekvivalent s while:

```
int i=0;
while (i < 10) {
    System.out.println(i);
    i++;
}
```

Výpis argumentů programu

```
public class Pole {
    public static void main(String[] args) {
        for(int i=0; i<args.length; i++)
            System.out.println(args[i]);
    }
}
```

Vícecestné větvení "switch - case - default"

- Obdoba pascalského select - case - else.
- Větvení do více možností na základě ordinální hodnoty

```
switch(výraz) {
    case hodnota1: prikaz1a;
                   prikaz1b;
                   break;
    case hodnota2: prikaz2a;
                   ...
                   break;
    default:       prikazDa;
                   ...
}
```

- Je-li výraz roven některé z hodnot, provede se sekvence uvedená za příslušným case.
- Sekvenci obvykle ukončujeme příkazem break, který předá řízení ("skočí") na první příkaz za ukončovací závorkou příkazu switch.

Příkaz "break"

- Realizuje "násilné" ukončení průchodu cyklem nebo větvením switch
- Syntaxe použití break v cyklu:

```
for (int i = 0; i < a.length; i++) {
    if(a[i] == 0) {
        break; // skoci se za konec cyklu
    }
}
if (a[i] == 0) {
    System.out.println("Nasli jsme 0 na pozici "+i);
} else {
    System.out.println("0 v poli neni");
}
```

Příkaz "continue"

- Používá se v těle cyklu.
- Způsobí přeskočení zbylé části průchodu tělem cyklu.
- Běh pokračuje další iterací.

```
for (int i = 0; i < a.length; i++) {
    if (a[i] == 5)
        continue;
    System.out.println(i);
}
```

Ladění programu

Pro ladění programů v Javě lze využít

- kontrolní tisky - System.err.println(...)
- řádkovým debuggerem jdb
- integrovaným debuggerem v IDE
- pomocí speciálních nástrojů na záznam běhu balíků

Uvědomte si, že žádný nástroj za nás nevymyslí, JAK máme své třídy testovat. Pouze nám pomůže ke snadnějšímu sestavení a spuštění testu.

Ladění programu

- standardní klíčové slovo (od JDK1.4) `assert`
  - `assert booleovský_výraz`

- testovací nástroje typu JUnit (a varianty - HttpUnit,...)
  - metoda `assertEquals()`
  - metoda `assertTrue()`
  - `...`
  - `http://junit.org/`

- pokročilé nástroje na běhovou kontrolu platnosti invariantů, vstupních, výstupních a dalších podmínek
  - např. jass (Java with ASSertions),
  - `http://csd.informatik.uni-oldenburg.de/~jass/`

Ladění programu - assert

```
public class Zlomek {
     int cit, jm;
     public Zlomek(int c, j) {
          assert j != 0;
          cit = c;
          jm = j;
     }
}
```

- Přeložit jej s volbou -source 1.4
- Spustit jej s volbou -ea (-enableassertions)
- Dojde-li za běhu programu k porušení podmínky stanovené za assert, vznikne běhová chyba (AssertionError) a program skončí.

Ladění programu - JUnit

Postup

- stáhnout si distribuci testovacího prostředí (stačí binární) – `http://junit.org`
- nainstalovat JUnit (tj. rozbalit do adresáře)
- napsat testovací třídu (třídy) – obvykle rozšiřují (dědí) třídu `junit.framework.TestCase`
- testovací třída obsahuje metodu na nastavení testu (`setUp`), testovací metody (`test`*`Neco`*) a úklidovou metodu (`tearDown`)
- testovací třídu spustit v textovém nebo grafickém prostředí – `junit.textui.TestRunner`, `junit.swingui.TestRunner`
- testování zobrazí, které testovací metody případně selhaly

Ladění programu - JUnit

```
public class JUnitDemo extends TestCase {
    Zlomek x, y;

    public void setUp() {
        x = new Zlomek(2,3);
        y = new Zlomek(4,6);
        z = new Zlomek(4,3);
    }

    public void testRovna() {
        assertEquals("2/3 a 4/6 se musi rovnat.",x,y);
    }

    public void testSoucet() {
        Zlomek z = x.plus(y);
        assertEquals("2/3 + 4/6 se musi rovnat 4/3.",
                                        z, soucet);
    }
}
```

Zadání 1. úkolu

- viz `http://www.fit.vutbr.cz/study/courses/IJA/public`