Seminář Java
III
2005/2006

Radek Kočí

Rekapitulace

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

Téma přednášky

- Balíky – přístup k třídám z jiných balíků
- Dědičnost tříd
- Inicializace objektu, konstruktory
- Řídicí operace
- Operátory
- Ladění programu

Přístup k třídám z jiných balíků

```
package seminar2.banka.ucty;
public class Ucet {
     ...
}
```

```
package seminar2.banka;
public class Banka {
     Ucet ucet = new Ucet();
     ...
}
```

```
 IJA
  |-- seminar2
           |-- banka
                  |-- Banka.java
                  |-- ucty
                        |-- Ucet.java
```

Přístup k třídám z jiných balíků

- tečková notace `seminar2.banka.ucty.Ucet`
- $\Rightarrow$ zdlouhavé, kompikované
- $\Rightarrow$ import tříd

```
package seminar2.banka.ucty;
public class Ucet {
     ...
}
```

```
package seminar2.banka;

public class Banka {
     seminar2.banka.ucty.Ucet ucet;
     ucet = new seminar2.banka.ucty.Ucet();
     ...
}
```

Import tříd z balíků

- klauzule `import` *`package.třída`*
- klauzule `import` *`package.*`*
- `*` nezpřístupní třídy z podbalíků!!

```
package seminar2.banka.ucty;
public class Ucet {
     ...
}
```

```
package seminar2.banka;
import seminar2.banka.ucty.Ucet;
public class Banka {
     Ucet ucet = new Ucet();
     ...
}
```

Import tříd z balíků – překlad

```
IJA
 |-- seminar2
          |-- banka
                   |-- Banka.java
                   |-- ucty
                            |-- Ucet.java
```

Př.: jsme v adresáři `IJA`

```
javac -classpath .  seminar2/banka/ucty/Ucet.java
javac -classpath .  seminar2/banka/Banka.java

java -classpath .  seminar2.banka.Banka
```

Import tříd z balíků

- balík `java.lang` je vždy importován automaticky
- třída `java.lang.System`

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;
    }
}
```

Příkazy v Javě

Co už známe ...

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

Nové ...

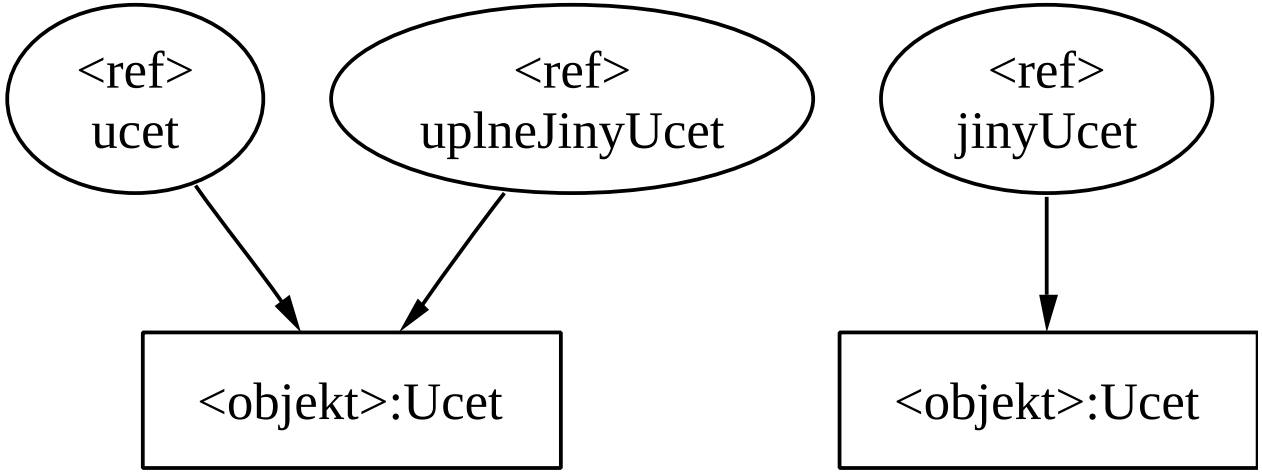
- přiřazovací příkaz (`=`)
- řízení toku programu

Přiřazení

- Na levé straně musí být proměnná.
- Na pravé straně musí být *přiřaditelný* výraz.
- Primitivní typy
  - přiřazením se hodnota zduplikuje
  - konverze typů (`short` $\rightarrow$ `int`, `int` $\rightarrow$ `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;
    }
}
```



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).
- Relativně málo používaný – je méně přehledný než `while`

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

Cyklus "for"

- 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++;
}
```

Vícecestné větvení "switch - case - default"

- 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);
}
```

Operátory a výrazy, porovnávání objektů

- Aritmetické
- Logické
- Relační
- Bitové
- Operátor podmíněného výrazu ? :
- Operátory typové konverze (přetypování)
- Operátor zřetězení +
- Relační operátory

Aritmetické operátory

- `+`, `-`, `*`, `/` a `%` (zbytek po celočíselném dělení)
- platí podobná pravidla jako v C/C++
  - `int / int` $\Rightarrow$ `int`
  - `double / int` $\Rightarrow$ `double`
  - `short / int` $\Rightarrow$ `int`

Logické operátory

- logické součiny (AND):
  - & (nepodmíněný - vždy se vyhodnotí oba operandy),
  - && (podmíněný - líné vyhodnocování - druhý operand se vyhodnotí, jen nelze-li o výsledku rozhodnout z hodnoty prvního)
- logické součty (OR):
  - | (nepodmíněný - vždy se vyhodnotí oba operandy),
  - || (podmíněný - líné vyhodnocování - druhý operand se vyhodnotí, jen nelze-li o výsledku rozhodnout z hodnoty prvního)
- negace (NOT):
  - !

Bitové operátory

Bitové:

- součin &
- součet |
- exkluzivní součet (XOR) ^ (znak "stříška")
- negace (bitwise-NOT) ~ (znak "tilda")

Posuny:

- vlevo << o stanovený počet bitů
- vpravo >> o stanovený počet bitů s respektováním znaménka
- vpravo >>> o stanovený počet bitů bez respektování znaménka

Operátor podmíněného výrazu ? :

Bitové:

- Jediný ternární operátor
- Platí-li první operand (má hodnotu true) $\Rightarrow$
  - výsledkem je hodnota druhého operandu
  - jinak je výsledkem hodnota třetího operandu
- Typ prvního operandu musí být boolean, typy druhého a třetího musí být přiřaditelné do výsledku.

```
if (a > b)
     c = a - b;
else
     c = b - a;
```

```
c = (a > b ? a - b : b - a);
```

Operátor zřetězení +

- Výsledkem je vždy řetězec, ale argumenty mohou být i jiných typů, např.
- následující sekvence je v pořádku
  `int i = 1; System.out.println("promenna i=" + i);`
- s řetězcovou konstantou se spojí řetězcová podoba dalších argumentů (např. čísla).
- Pokud je argumentem zřetězení odkaz na objekt o ⇒
  - je-li `o == null` ⇒ použije se řetězec `null`
  - je-li `o != null` ⇒ použije se hodnota vrácená metodou `o.toString()` (tu lze překrýt a dosáhnout tak očekávaného řetězcového výstupu)

Relační (porovnávací) operátory

- Tyto lze použít na porovnávání primitivních hodnot:
  - `<`, `<=`, `>=`, `>`
- Test na rovnost/nerovnost lze použít na porovnávání primitivních hodnot i objektů:
  - `==`, `!=`
  - pozor na srovnávání floating-points čísel na rovnost: je třeba počítat s chybami zaokrouhlení; místo porovnání na přesnou rovnost raději používejme jistou toleranci: `abs(expected-actual) < delta`
  - pozor na porovnávání objektů: `==` vrací `true` jen při rovnosti odkazů, tj. jsou-li objekty identické!

Operátory typové konverze (přetypování)

- Podobně jako v C/C++
- Píše se `( typ )` hodnota
- např. `(short) o`, kde `o` byla proměnná deklarovaná jako `int`.

- u primitivních typů se jedná o úpravu hodnoty – např. `int` přetypujeme na `short` a "ořeže" se tím rozsah.

Ladění programu

Pro ladění programů v Javě lze využít

- kontrolní tisky: `System.err.println(...)`
- řádkový debugger `jdb`
- integrovaný debugger v IDE
- speciální nástroje 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 AssertDemo {
    public static void main(String args[]) {
        int x = 10;
        boolean enabled = false;
        assert enabled = true;
        System.out.println("Assertions are " +
                           (enabled ? "enabled" : "disabled");
        assert x < 0 :  "x is not < 0";
    }
}
```

- přeložit s volbou `-source 1.4`
- spustit 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 metody
  - metodu pro nastavení testu – `setUp()`
  - testovací metody – `test`*`Neco`*`()`
  - 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, z;

    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);
    }
}
```