Seminář Java

II

Radek Kočí

Fakulta informačních technologií VUT

14. ledna 2007

Téma přednášky

- Pole
- Příkazy, operátory
- Deklarace třídy, konstruktory
- Dědičnost, inicializace obejktů
- Ladící nástroje

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

- `typ [] 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 typ[velikost];`
  - `int[] pole = new int[10];`
- nebo inicializací při deklaraci:
  - `int[] nazevPole = { 1, 2, 3 };`

Syntaxe přístupu k prvkům

- `pole[i]`
- `pole[i] = 20;`
- `int j = pole[2];`

Pole – III

```
Ucet [] ucty;                  // deklarace pole
ucty = new Ucet[5];            // vytvoření pole

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

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`.

Kopírování pole

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

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

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]);
    }
}
```

Příkazy v Javě

Co už známe ...

- volání metody
- příkaz je ukončen středníkem (`;`)

Nové ...

- přiřazovací příkaz (`=`)
- řízení toku programu
- návrat z metody (`return`)

Přiřazení

Přiřazení =

- Na levé straně musí být proměnná.
- Na pravé straně musí být *přiřaditelný* výraz.

Primitivní typy

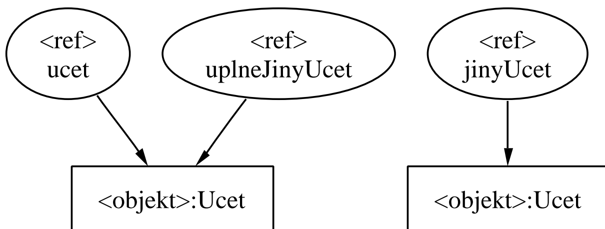
- přiřazením se hodnota zduplikuje
- konverze typů (`short` → `int`, `int` → `short`)

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;
    }
}
```



Řídicí 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;
   ...
}
```

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

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;
    ...
} while (podmínka);
```

Cyklus "for"

- de-facto jde o rozšíření while, lze jím snadno nahradit

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

Příklad použití "for" cyklu

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

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("0 je 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

- Aritmetické
- Logické
- Relační
- Bitové
- Operátor podmíněného výrazu ? :
- Relační operátory

Aritmetické operátory

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

Podmíněný výraz

- 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);
```

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!

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)

Příklad

Chceme modelovat následující skutečnost

- máme dopravce vlastnící dopravní prostředky
- dopravce má zaměstnance, kteří mohou řídit dopravní prostředky
- každý prostředek může být řízen některým ze zaměstnanců

Příklad

```
public class Car {
    protected Driver drivenBy;
    public void DrivenBy(Driver d) {
        drivenBy = d;
    }
}

public class Driver {
    protected String name;
    protected Car driveCar;
    public void driveCar(Car c) {
        driveCar = c;
        driveCar.drivenBy(this);
    }
}
```

Deklarace třídy

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

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

Základní pojmy – Atributy objektu

Atribut vs. proměnná objektu

- reprezentují data zapouzdřená v objektu
- *Proměnná objektu*
  - implementační pohled
  - získání/nastavení atributu ⇒ lze (teoreticky) přímo (nedoporučuje se)
- *Atribut objektu*
  - pohled z vyšší úrovně
  - atribut je vlastnost objektu
  - atribut není proměnná (i když je tak většinou realizován)
  - atribut datum (dd/mm/rr) ⇒ počet sekund od LP 1970
  - atribut objem ⇒ součin tří hodnot
  - získání/nastavení atributu ⇒ operace

Deklarace proměnné objektu

Inicializace proměnné objektu (primitivní datový typ)

- **`int capacity;`**
- *capacity* `== 0`

Inicializace proměnné objektu (objektový typ)

- **`Driver drivenBy;`**
- *drivenBy* `== null`

Deklarace metody

```
modifikátory typ název ( 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

Deklarace proměnné v metodě

- deklarace bez modifikátorů
- nemá implicitní inicializaci

```
public void run(String name, Car car) {
   Driver d = collection.getDriver(name);
   d.driveCar(car);
}
```

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`

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

Příklad

```
public class Car {
    protected int weight;
    protected int capacity;
    protected Driver drivenBy;

    public void DrivenBy(Driver d) {
        drivenBy = d;
    }
    public int getWeight() {
        return weight;
    }
    public int getCapacity() {
        return capacity;
    }
}
```

Vytváření objektů

Voláním např. `new Car()` 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 Car {
    public Car() {
        ...
    }
}
```

Použití: `new Car();`

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 Car {
    public Car(int w) {
        weight = w;
    }
}
```

Použití:

```
new Car(5000);
new Car();        ⇐ chyba!
```

Další konstruktory

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

```
public class Car {
    public Car() { }
    public Car(int w) {
        weight = w;
    }
}
```

Použití:

```
new Car(5000);
new Car();
```

⇐ OK

Příklad

```
public class Car {
    protected int weight;
    protected int capacity;
    protected Driver drivenBy;
    protected int km;

    public Car(int weight, int capacity) { ... }
    public int run(int km) {
        this.km += km;
        return price(km);
    }
    protected int price(int km) {
        // podle typu vozidla ...
    }
}
```

Příklad

```
public class Car2 {
    protected int weight;
    protected int capacity;
    protected Driver drivenBy;
    protected int km;

    public Car(int weight, int capacity) { ... }
    public int run(int km) {
        this.km += km;
        return price(km);
    }
    protected int price(int km) {
        // podle typu vozidla ...
    }
}
```

Základní pojmy OO – Dědičnost

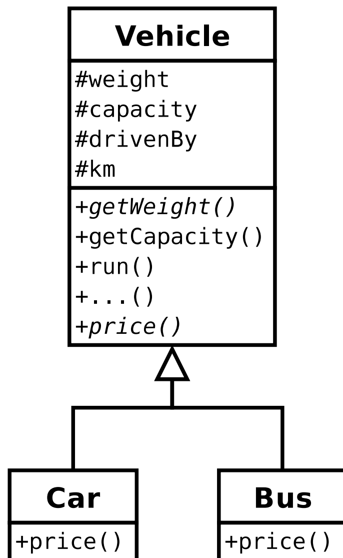
Dědičnost

- vyjadřuje hierarchický vztah mezi objekty
- definuje a vytváří objekty na základě již existujících objektů
    - možnost sdílení chování bez nutnosti reimplementace
    - možnost rozšíření chování

Způsob vyjádření dědičnosti závisí na typu jazyku

- třídně založené jazyky (Java) $\Rightarrow$ dědičnost tříd

Dědičnost (třídně založené jazyky)



Přepisování a přetěžování metod

Přepisování (overriding)

- změna definice metody zadané v třídě T v některé z podřízených tříd

Přetěžování (overloading)

- technika vícenásobné definice operace v jedné třídě.
- Java:
```
prevedNa(Ucet u, int castka);
prevedNa(Ucet u);
```
- Smalltalk nezná přetěžování:
```
preved:  castka na:  u.
prevedNa:  u.
```

Příklad

```
public class Vehicle {
    ...
}

public class Car extends Vehicle {
    protected int price(int km) {
        // podle osobniho auta ...
    }
}

public class Bus extends Vehicle {
    protected int price(int km) {
        // podle autobusu ...
    }
}
```

Dědičnost

- *Sdílení chování.*
- 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!!!*

Příklad

```
public class Vehicle {
    ...
    public Vehicle(int weight, int capacity) {
        this.weight = weight;
        this.capacity = capacity;
    }
}

public class Car extends Vehicle {
    // Funkcni, ovsem nevhodne (viz inic. obj.)!!
    public Car(int weight, int capacity) {
        this.weight = weight;
        this.capacity = capacity;
    }
    ...
}
```

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("Kon.  Z");}
}

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

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

Inicializace objektu – příklad

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

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

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

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

Inicializace objektu – příklad

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

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

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

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

Inicializace objektu – příklad

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

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

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

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

Kon. A

Inicializace objektu – příklad

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

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

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

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

```
Kon.  A
```

Inicializace objektu – příklad

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

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

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

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

Kon. A – **Kon. Z**

Inicializace objektu – příklad

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

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

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

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

```
Kon.  A  –  Kon.  Z  –  Kon.  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

Příklad

```
public class Vehicle {
    ...
    public Vehicle(int weight, int capacity) {
        this.weight = weight;
        this.capacity = capacity;
    }
}

public class Car extends Vehicle {
    public Car() {
        super(2000, 5);
    }
    ...
}
```

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";
    }
}
```

Ladění programu – assert

- 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

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

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

Přístup k třídám z jiných balíků

- tečková notace `junit.framework.TestCase`
- $\Rightarrow$ zdlouhavé, kompikované
- $\Rightarrow$ import tříd

```
package junit.framework;
public class TestCase { ...  }
```

```
package homework1;
public class Test
       extends junit.framework.TestCase {
   homework1.histogram.Histogram h;
   h = new homework1.histogram.Histogram();
   ...
}
```

Import tříd z balíků

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

```
package homework1;
import junit.framework.TestCase;

public class Test extends TestCase {
    ...
}
```

Import tříd z balíků

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