Seminář Java
III

Radek Kočí

Fakulta informačních technologií VUT

Únor 2011

Téma přednášky

- Dědičnost, polymorfismus
- Inicializace objektu
- Nástroje
  - java archiv (JAR)
  - ant
  - ladění programu (JUnit)

Základy objektové orientace

Vlastnosti objektové orientace

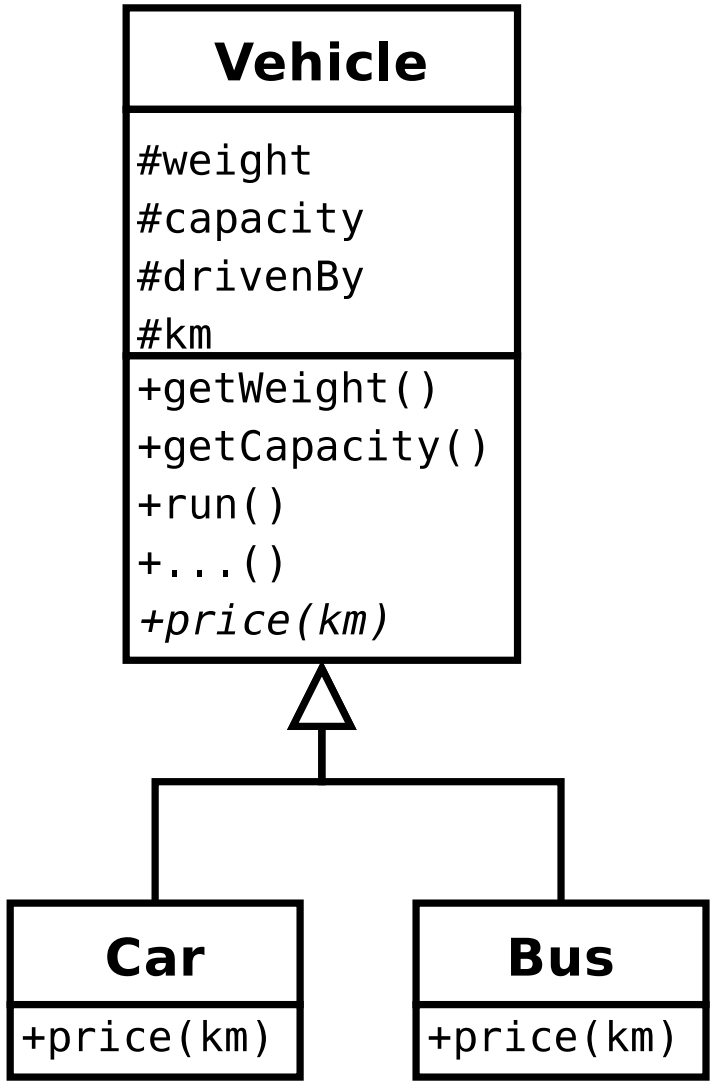
- *Abstrakce (abstraction)*
- *Zapouzdření (encapsulation)*
- Polymorfismus (polymorphism)
  - možnost vícenásobné definice operace s jedním názvem, která tak může nabývat více implementací (implementuje různé chování)
- Dědičnost (inheritance)
  - sdílení chování

Dědičnost v Javě

Dědičnost

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

Dědičnost v Javě – Příklad



Dědičnost v Javě – 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 ...
    }
}
```

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);
```

Dědičnost – Konstruktory (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 – modifikace

Možné modifikace

- lze volat jiný než bezparametrický konstruktor nadřazené třídy (musí být vždy na začátku konstruktoru potomka), např.

  ```
  super(parametry)
  ```

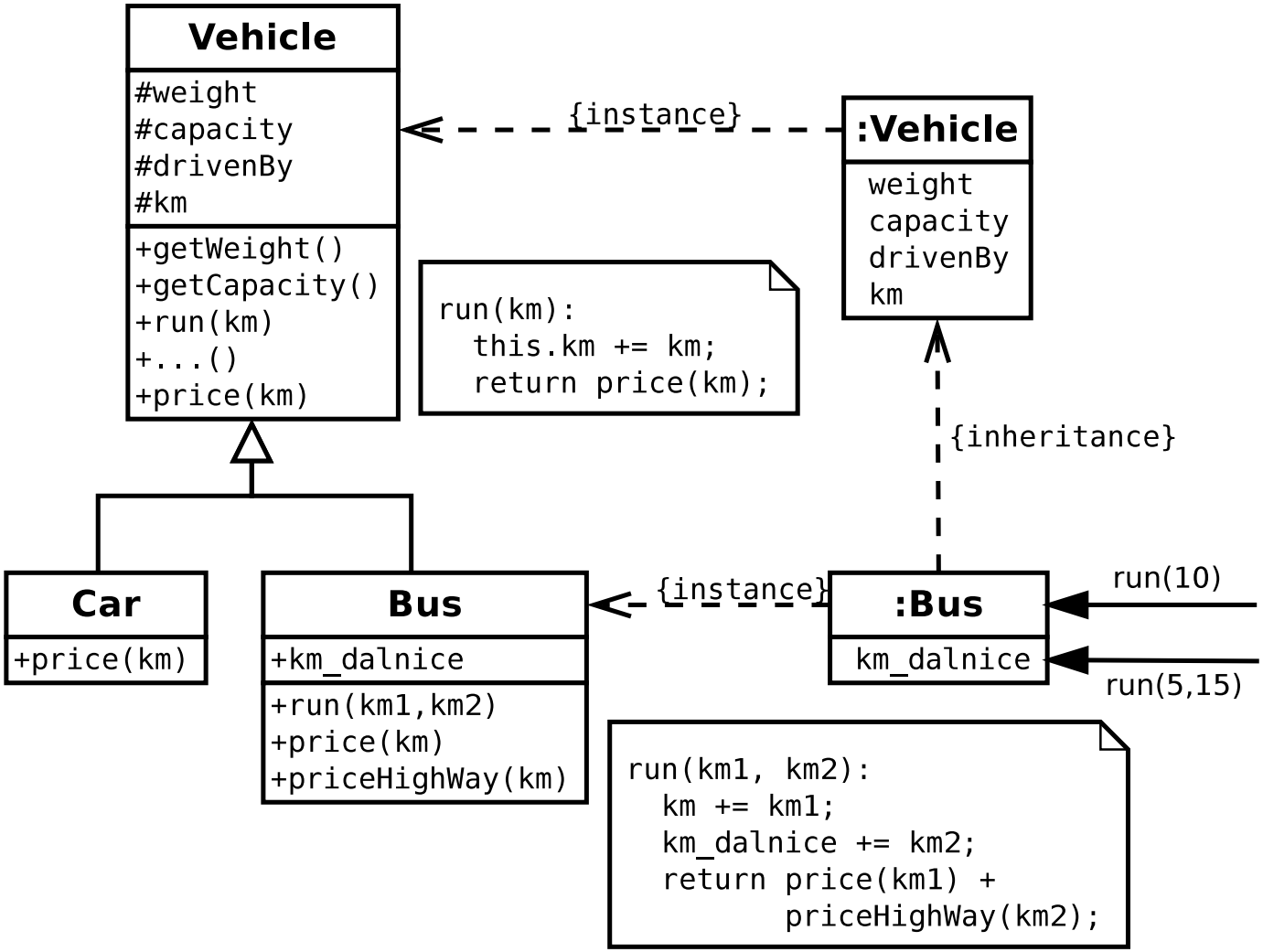
- lze volat i jiný konstruktor třídy (musí být vždy na začátku konstruktoru), např.

  ```
  this(parametry)
  ```

- *bezparametrický (implicitní) konstruktor neexistuje, pokud existuje alespoň jeden jiný*

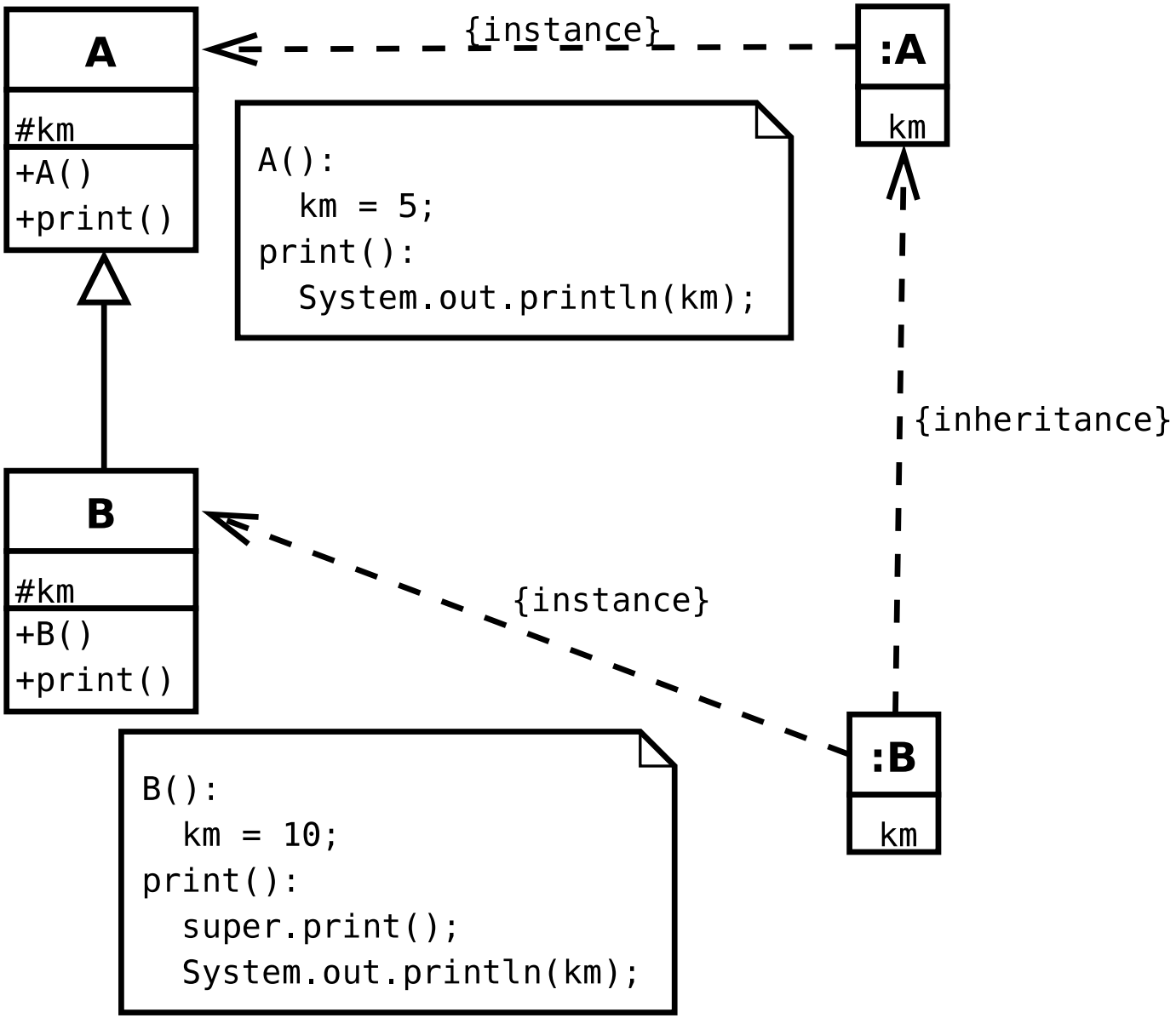
  `super` a `this` *lze použít i pro volání metod nadřazené/dané třídy*

Dědičnost – Vztah objektů a tříd



Poznámka: polymorfní metody

Dědičnost – vztah objektů a tříd



Dědičnost – Konstruktory (příklad)

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

public class Car extends Vehicle {
    public Car(int weight, int capacity) {
        super(weight, capacity);
    }
    ...
}
```

Dědičnost versus skládání

Příklad

- úložiště objektů třída `U`
- metody `put(...)`, `get(...)`, `remove(...)`
- chceme naplnit a pak prohlásit za *read-only*

Dědičnost

- zdědíme třídu `U`
- atribut `isReadOnly`
- modifikace metod `put(...)` a `remove(...)`

Dědičnost versus skládání

```
class U { ... }

class UU extends U {
    protected boolean isReadOnly = false;

    public void readOnly(boolean ro) {
        isReadOnly = ro;
    }

    public void put(Object o) {
        if (! isReadOnly)
            return super.put(o);
        else
            ...
    }
}
```

Dědičnost versus skládání

Skládání

- vytvoříme *jinou* třídu **`RU`**
- skládá se z třídy (resp. instance třídy) **`U`**
- deleguje zprávy na složkový objekt (**`get(...)`**)
- metody **`put(...)`** a **`remove(...)`** buď neimplementuje, nebo vždy generuje výjimku

Po vytvoření a naplnění instance třídy **`U`**

- vytvoříme instanci třídy **`RU`** a vložíme do ní inicializovanou instanci třídy **`U`**

Dědičnost versus skládání

```
class U { ... }

class RU {
    protected U inner;

    public RU(U u) {
        inner = u;
    }

    public Object get() {
        return inner.get();
    }
}
```

Dědičnost – problém narušení zapouzdření

```
class MyHashSet extends HashSet {
    private int addCount = 0;
    // konstruktory
    public boolean add(Object o) {
        addCount++;
        return super.add(o);
    }
    public boolean addAll(Collection c) {
        addCount += c.size();
        return super.addAll(c);
    }
    public int getCount() {
        return addCount;
    }
}
```

Dědičnost – problém narušení zapouzdření

```
MyHashSet s = new MyHashSet();
s.addAll(Arrays.asList(
    new String[] {"jedna", "dva", "tri"} ));

s.getCount();           // => 6
```

- metoda `addAll` třídy `HashSet` používá metodu `add`
- implementační detail, který nemusí být dokumentovaný

Dědičnost – problém narušení zapouzdření

```
MyHashSet s = new MyHashSet();
s.addAll(Arrays.asList(
    new String[] {"jedna", "dva", "tri"} ));

s.getCount();          // => 6
```

- metoda `addAll` třídy `HashSet` používá metodu `add`
- implementační detail, který nemusí být dokumentovaný

Dědičnost – problém narušení zapouzdření

Dědičnost

- sémantika je založena na implementačních detailech rozšiřované třídy $\Rightarrow$ náchylné na chyby

```
// class Vehicle
public Vehicle(int weight) {
    this.weight = weight * 1000;
}

// class Car extends Vehicle
public Car(int weight) {
    this.weight = weight; // <-- !!!
    // super(weight);
}
```

Nástroje

- java archiv (JAR)
- ant
- ladění programu (JUnit)

Java Archive (JAR)

- platformově nezávislý
- archiv (zip) obsahující
  - hierarchii balíků a class soubory
  - jakékoliv jiné soubory (obrázky pro aplety apod.)
  - speciální adresář `META-INF`

`META-INF`

- obsah je interpretován JVM
- konfigurace aplikace
- konfigurace rozšíření
- konfigurace zavaděčů tříd a služeb

Zkompilování zdrojových textů

```
-- src
    |---- xml
            |---- XMLDemo.java
-- dest
-- dom4j-1.5.2.jar
```

```
javac -classpath "src:dom4j-1.5.2.jar"
              -d dest src/xml/XMLDemo.java
```

Zkompilovani zdrojovych textu

```
-- src
    |---- xml
            |---- XMLDemo.java
-- dest
    |---- xml
            |---- XMLDemo.class
-- dom4j-1.5.2.jar
```

Java Archive (JAR)

```
-- dest
    |---- xml
            |---- XMLDemo.class
-- dom4j-1.5.2.jar
```

```
jar -cvf xml.jar -C dest xml
```

MANIFEST.MF:

```
Manifest-Version: 1.0
Created-By: 1.5.0_05 (Sun Microsystems Inc.)
```

Java Archive (JAR)

```
-- dest
    |---- xml
            |---- XMLDemo.class
-- dom4j-1.5.2.jar
```

```
jar -cvfm xml.jar mymanifest.mf -C dest xml
```

```
MANIFEST.MF:
Manifest-Version: 1.0
Class-Path: dom4j-1.5.2.jar
Created-By: 1.5.0_05-b05 (Sun Microsystems Inc.)
Ant-Version: Apache Ant 1.6.2
Main-Class: xml.XMLDemo
```

Java Archive (JAR)

Spuštění JAR souboru

```
java -jar xml.jar
```

---

http://java.sun.com/j2se/1.5.0/docs/guide/jar/

Apache Ant

Apache Ant

- nástroj pro sestavování aplikací
- v principu podobný nástroji `make`
- sestavovací soubory založeny na XML formátu

Sestavovací soubor (buildfile)

- projekt (project)
- cíle (targets)
- úlohy (tasks)
- závislosti

`http://ant.apache.org`

Soubor `build.xml`

```
<project>
  <target name="clean">
    <delete dir="build"/>
  </target>

  <target name="compile">
    <mkdir dir="build/classes"/>
      <javac srcdir="src" destdir="build/classes"/>
  </target>
```

Apache Ant

```
    <target name="jar">
        <mkdir dir="build/jar"/>
        <jar destfile="build/jar/xml.jar"
                        basedir="build/classes">
            <manifest>
                <attribute name="Main-Class"
                        value="xml.XMLDemo"/>
            </manifest>
        </jar>
    </target>

    <target name="run">
        <java jar="build/jar/xml.jar" fork="true"/>
    </target>
</project>
```

Apache Ant

```
ant compile
ant jar
ant run
```

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

- 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

Instalace JUnit

- stáhnout si distribuci testovacího prostředí (stačí binární)
  `http://junit.org`
- nainstalovat (rozbalit do adresáře) $\rightarrow$ archiv jar

Spuštění testů

- `javac -cp junit.jar ...`

Ladění programu – JUnit

Postup (starší verze)

- napsat testovací třídu (třídy) – obvykle rozšiřují (dědí) třídu `junit.framework.TestCase`
- testovací třída obsahuje metody
  - metodu pro nastavení testu – `setUp()`
  - testovací metody – `testNeco()`
  - úklidovou metodu – `tearDown()`
- testovací třídu spustit v textovém nebo grafickém prostředí
  - `junit.textui.TestRunner`
  - `junit.swingui.TestRunner`
- testování zobrazí, které testovací metody případně selhaly

Ukázka (starší verze)

```
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);
    }
}
```

Ladění programu – JUnit

Využití anotací (annotation)

- anotace definují dodatečné informace a data o programu bez přímého vlivu na program
- využití anotací
  - při kompilaci – detekce chyb, možnost generování dalšího kódu, ...
  - za běhu – nástroje a knihovny mohou na základě anotace přizpůsobit sémantiku

Static import

Přístup ke statickým členům

- `double r = Math.cos(Math.PI * theta);`
- Využití statického importu

```
import static java.lang.Math.PI;
//import static java.lang.Math.*;
...

double r = cos(PI * theta);
```

- používat velice opatrně! (kolize identifikátorů, těžko čitelný kód, ...)

Ukázka využití anotací (annotation)

```
import org.junit.*;
import static org.junit.Assert.*;

public class TestHW {
    @Test public void xxx() {
        ...
        assertTrue(x>10);
    }
```

```
java org.junit.runner.JUnitCore homework1.TestHW
```