Seminář Java
III

Radek Kočí

Fakulta informačních technologií VUT

Únor 2012

Téma přednášky

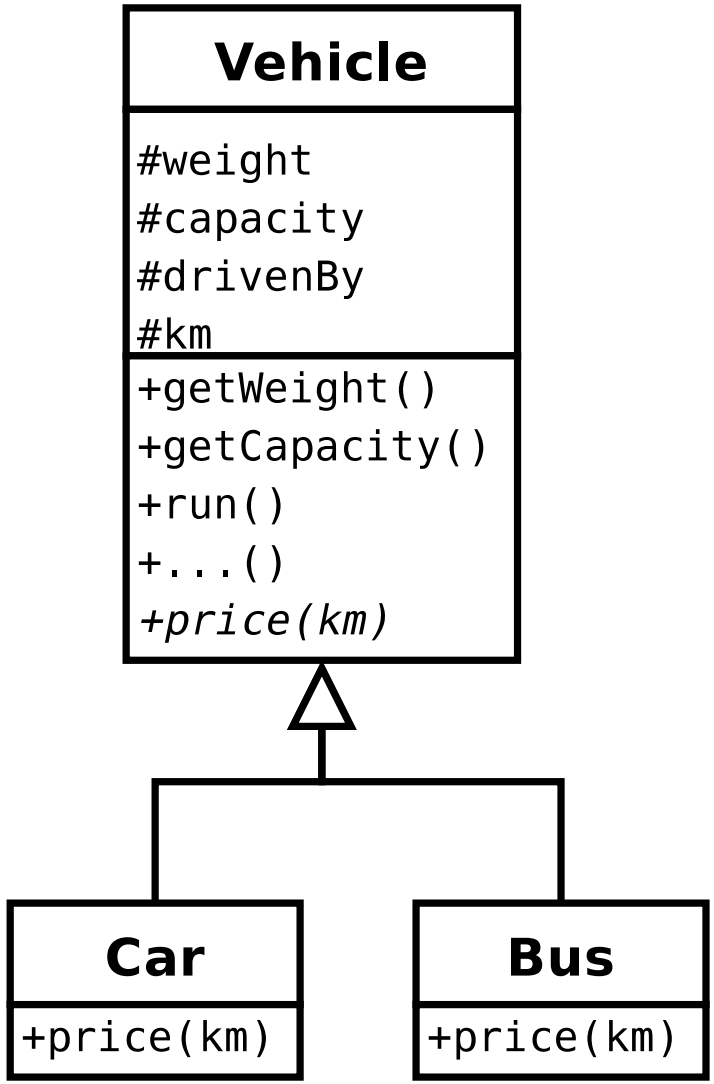
- Dědičnost, polymorfismus
- Inicializace objektu
- Hierarchie dědičnosti – třída `Object`
- Typová konverze, porovnávání objektů

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)
  ```

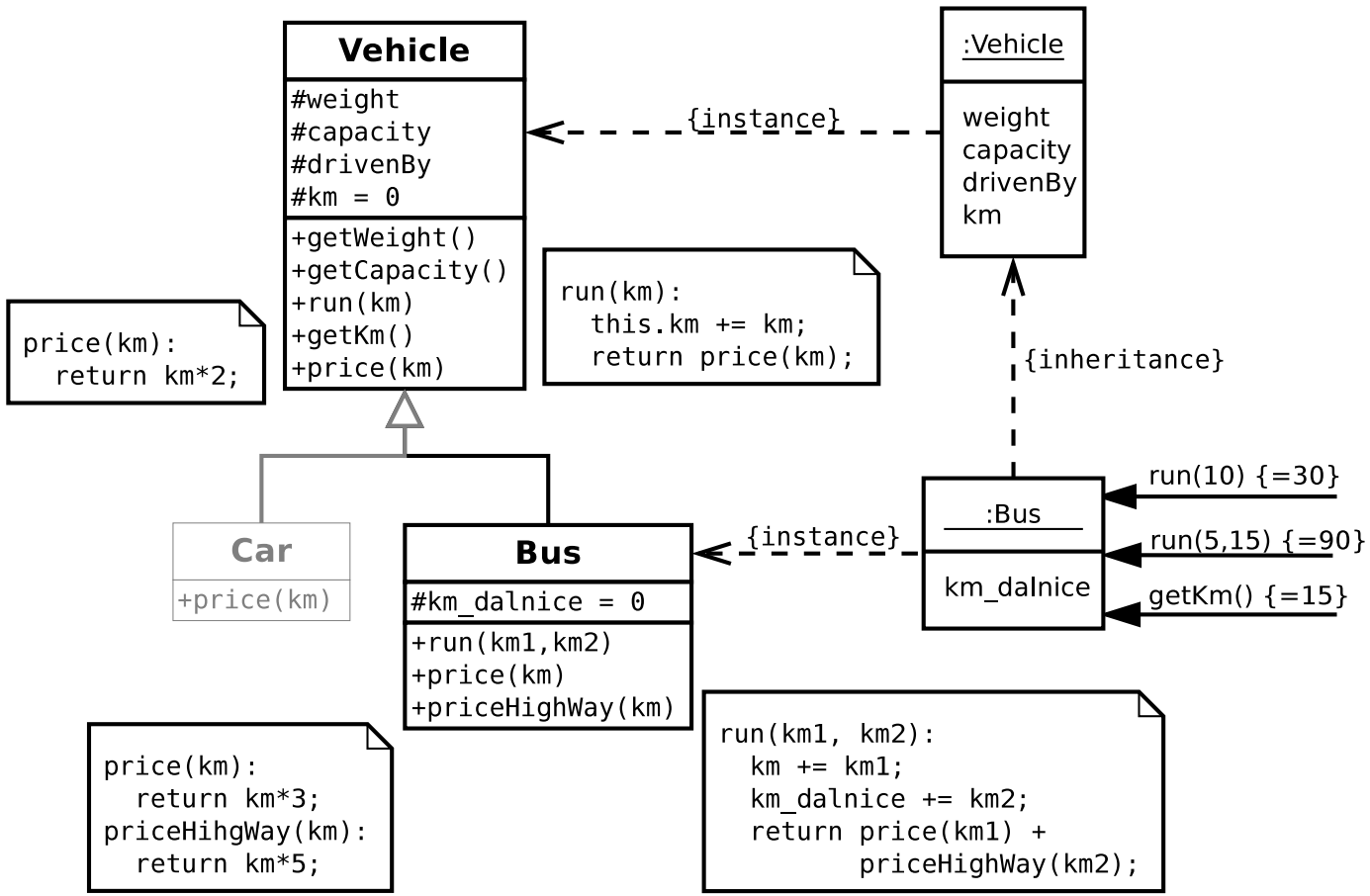
- lze volat i jiný konstruktor třídy (musí být vždy na začátku konstruktoru), např.

  ```
  this(parametry)
  ```

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

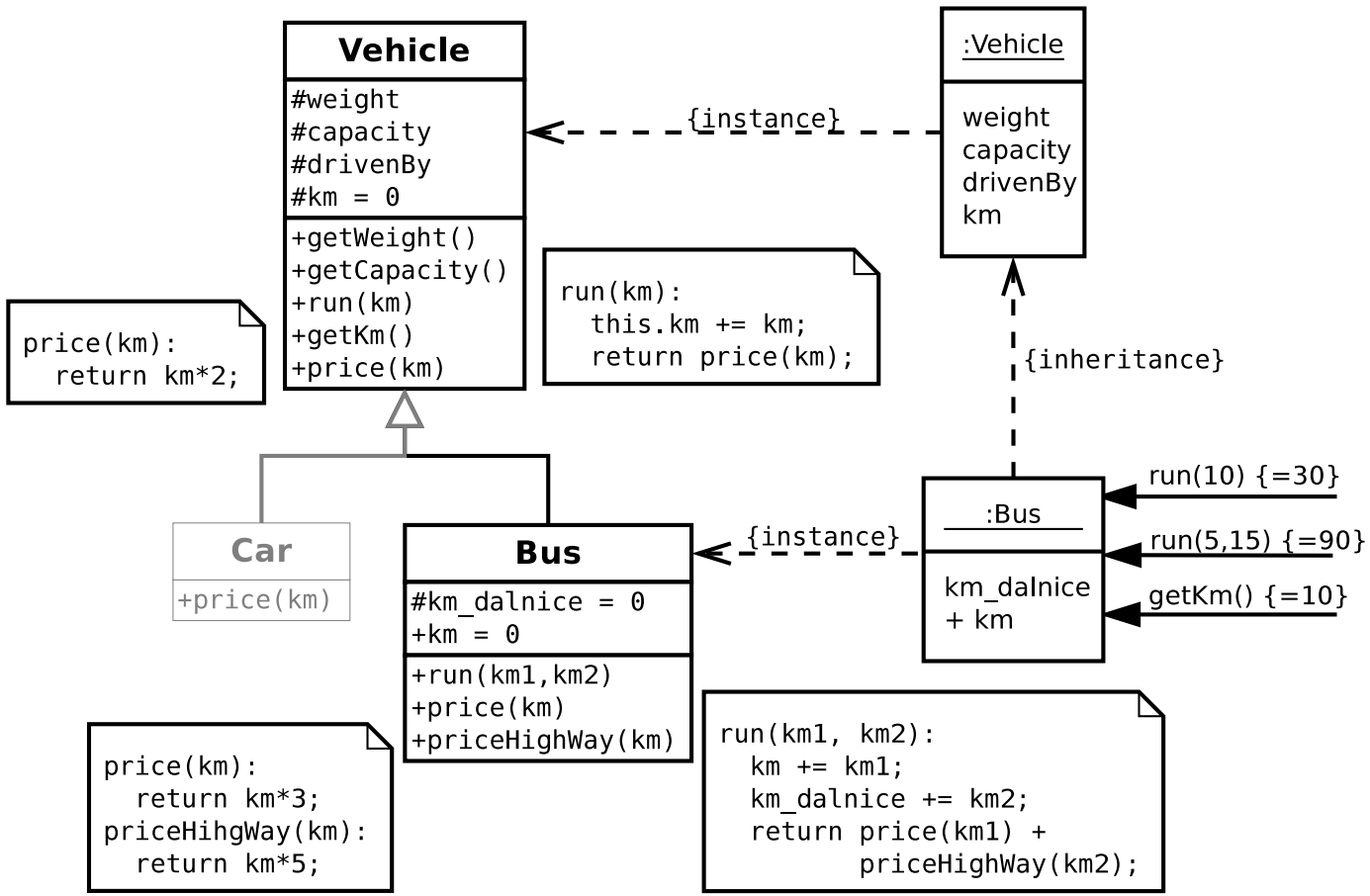
  `super` a `this` *lze použít i pro volání metod nadřazené/dané třídy*

Dědičnost – vztah objektů a tříd



Poznámka: polymorfní metody (`price(km)`)

Dědičnost – vztah objektů a tříd



Chyba v implementaci třídy `Bus` – objektová proměnná `km`

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;

    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

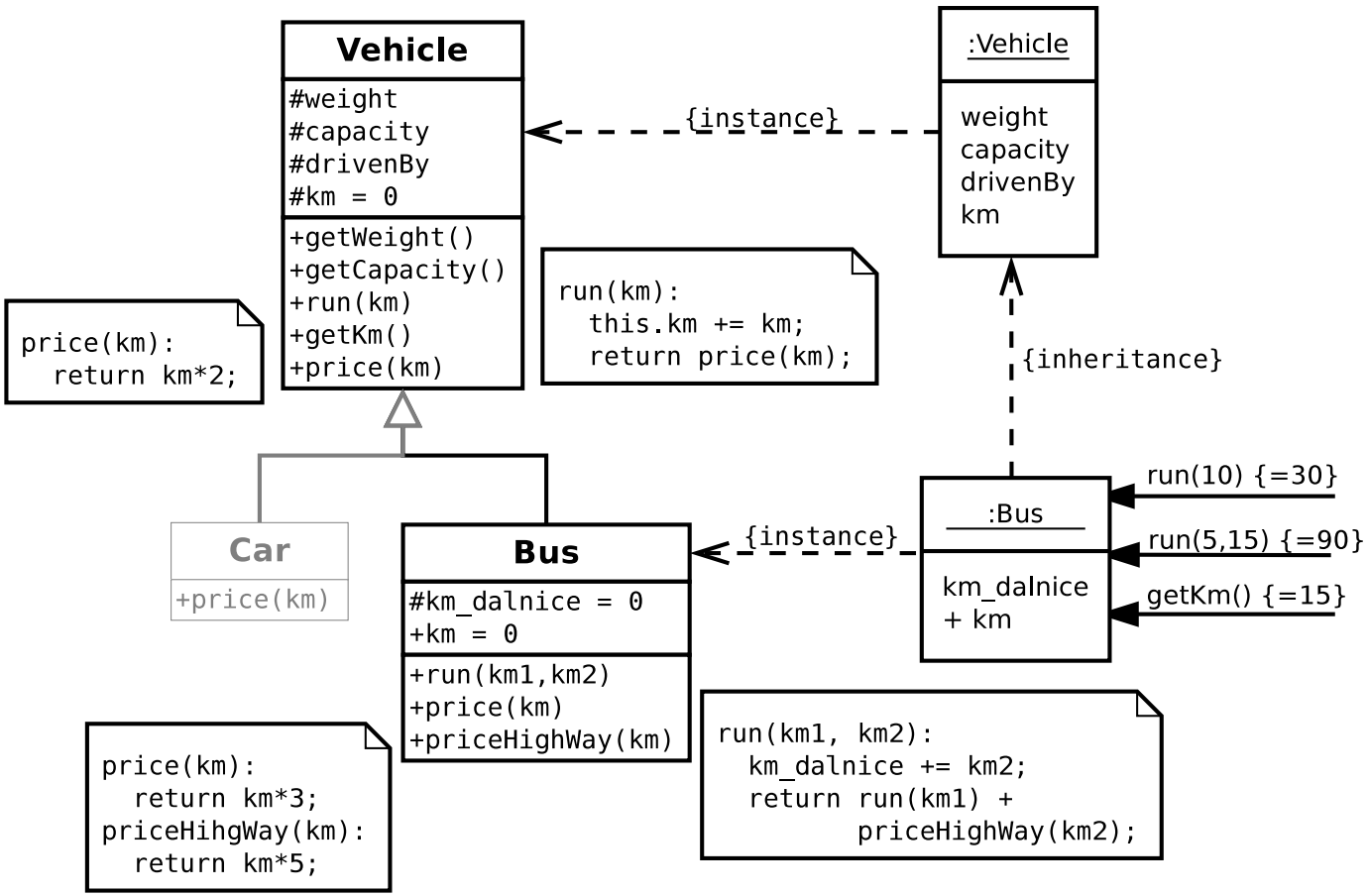
Dědičnost

- 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);
}
```

Dědičnost – problém narušení zapouzdření



Bezpečnější implementace metody `run(km1,km2)`.

Potenciální problém s překrytím objektové proměnné `km` zůstává

Třída Object

- Třída `Object` je předkem všech tříd.
- Definuje základní množinu operací
  - `public boolean equals(Object obj);`
  - `public int hashCode();`
  - `public String toString();`
- Do proměnné, jejíž typ je deklarován jako třída `A`, lze dosadit všechny instance třídy `A` a všechny instance tříd odvozených od třídy `A`.

Operátor zřetězení +

- Výsledkem je vždy řetězec, ale argumenty mohou být i jiných typů, např.
- např.
  `System.out.println("proměnná o = " + o);`
  - je-li `o` primitivní datový typ ⇒ přetypuje se na řetězec
  - 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)

Operátor typové konverze (přetypování)

Operátor typové konverze

- `( typ ) hodnota`

Typová konverze primitivních datových typů

- např. `(int) o`, kde `o` byla proměnná deklarovaná jako `long`.
- s konverzí primitivního typu může dojít ke změně hodnoty

Typová konverze objektových typů

- např. `(Ucet) o`, kde `o` byla proměnná deklarovaná jako `Object`.
- pro objektové typy se ve skutečnosti nejedná o žádnou konverzi spojenou se změnou obsahu objektu, nýbrž pouze o potvrzení, že běhový typ objektu je požadovaného typu – např. (viz výše) že `o` je typu `Ucet`.

Porovnávání objektů

Porovnávání objektů prostřednictvím operátoru `==` (`!=`)

- `true` ⇒ jedná se o dva odkazy na tentýž objekt – tj. o dva totožné objekty
- `false` ⇒ jedná se o dva odkazy na různé samostatné objekty – mohou být i stejné třídy i se stejným obsahem
- test identity (totožnosti)

Porovnávání objektů na základě jejich obsahu (tedy ne podle referencí)

- tj. dva objekty jsou rovné (rovnocenné, nikoli totožné), mají-li stejný obsah
- metoda `equals(Object o)`
- test rovnocenosti

Porovnávání objektů

Metoda `equals`

- je deklarovaná ve třídě `Object` (tj. každý objekt má metodu `equals`)
- *tato metoda (ve třídě `Object`) funguje přísným způsobem, tj. rovné si budou jen totožné objekty!*

Chceme-li chápat rovnost objektů podle obsahu

- musíme pro danou třídu překrýt metodu `equals`, která musí vrátit `true`, právě když se obsah výchozího a srovnávaného objektu rovná

Operátor instanceof

Operátor porovnání typů `instanceof` testuje

- zda je objekt instancí dané třídy
- zda je objekt instancí potomka dané třídy
- zda je objekt instancí třídy, která implementuje dané rozhraní

Porovnávání objektů – příklad

Dva objekty třídy `Ucet` jsou shodné, mají-li stejného majitele a zůstatek.

```
public class Ucet {
  protected String majitel;
  protected double zustatek;
  public Ucet (String jmeno) {
    majitel = jmeno;
  }
  ...
```

Porovnávání objektů – příklad

```
    ...
    public boolean equals(Object o) {
      if (o instanceof Ucet)  {
        Ucet c = (Ucet)o;
        return (zustatek == c.zustatek ?
          majitel.equals(c.majitel):  false);
      } else
        return false;
      }
    }
}
```

Vlastnosti metody equals

`equals`

- reflexivní:
  `x.equals(x) == true`
- symetrická:
  `x.equals(y) == true` $\Rightarrow$ `y.equals(x) == true`
- tranzitivní:
  `x.equals(y) == true` a `y.equals(z) == true` $\Rightarrow$
  `x.equals(z) == true`
- `x.equals(null) == false`

Symetrie metody `equals`

```
class CaseInsensitiveString {
   String s;
   public boolean equals(Object o) {
      if (o instanceof CaseInsensitiveString) ...
      if (o instanceof String)
         return s.equalsIgnoreCase((String) o);
      return false;
   }
}
```

```
CaseInsensitiveString cis =
          new CaseInsensitiveString("Pp");
String s = "pp";
cis.equals(s);  // true
s.equals(cis);  // false
```

Metoda `hashCode`

Jakmile u třídy překryjeme metodu `equals`, měli bychom současně překrýt i metodu `hashCode()`:

- `hashCode` vrací celé číslo (`int`) ”co nejlépe” charakterizující obsah objektu
- pro dva stejné (`equals`) objekty musí *vždy vrátit stejnou hodnotu*
- pro dva obsahově různé objekty by `hashCode` naopak měl vracet různé hodnoty (ale není to stoprocentně nezbytné a ani nemůže být vždy splněno)

Metoda hashCode - příklad

V těle `hashCode` často delegujeme řešení na volání `hashCode` jednotlivých složek objektu – a to těch, které figurují v `equals`:

```
public class Ucet {
  protected String majitel;
  protected double zustatek;
  public Ucet (String jmeno) {
    majitel = jmeno;
  }
  public boolean equals(Object o) { ... }
  public int hashCode() {
    return majitel.hashCode();
  }
}
```

Několik poznámek

Ladění programu – assert

- standardní klíčové slovo (od JDK1.4) `assert`
  - `assert booleovský_výraz`
- 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 – 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";
    }
}
```

Reflektivní vlastnosti

Reflektivita

- zkoumání tříd a objektů
- speciální objekty reprezentující vlastnosti tříd a objektů
- každý element (třída, metoda, ...) má svou reprezentaci v podobě objektu
- `java.lang.Class<T>`
- `java.lang.reflect.Constructor<T>`
- `java.lang.reflect.Method`
- `java.lang.reflect.Field`

Reflektivní vlastnosti

Třída

- každá zkompilovaná třída (bytecode) má proměnnou:
  `static public final class`
  př.:
  `java.lang.Class cls = Myclass.class;`
- metoda `Object.getClass()`

```
String str = new String("Hi");
Class cls = str.getClass();
```

Reflektivita – získání jména třídy objektu

```
import java.lang.reflect.*;
import java.awt.*;

class SampleName {

   public static void main(String[] args) {
      Button b = new Button();
      printName(b);
   }

   static void printName(Object o) {
      Class c = o.getClass();
      String s = c.getName();
      System.out.println(s);
   }
}
```

Reflektivita – nadřazené třídy

```
static void printSuperclasses(Object o) {
  Class subclass = o.getClass();
  Class superclass = subclass.getSuperclass();

  while (superclass != null) {
    String className = superclass.getName();
    System.out.println(className);
    subclass = superclass;
    superclass = subclass.getSuperclass();
  }
}
```

Reflektivita – proměnné, modifikátory

```
static void printFieldNames(Object o) {
  Class c = o.getClass();
  Field[] fields = c.getDeclaredFields();

  for (int i = 0; i < fields.length; i++) {
    String fieldName = fields[i].getName();
    Class typeClass = fields[i].getType();
    String fieldType = typeClass.getName();

    int modif = fields[i].getModifiers();
    if (Modifier.isPublic(modif)) ...
    if (Modifier.isPrivate(modif)) ...
    if (Modifier.isStatic(modif)) ...
    ...
  }
}
```