

Seminář Java

Základy objektové orientace II

Radek Kočí

Fakulta informačních technologií VUT

Únor 2008

- Dědičnost, polymorfismus
- Inicializace objektu
- Rozhraní, třída a objekt v Javě

Vlastnosti objektové orientace

- *Abstrakce (abstraction)*
- *Zapouzdření (encapsulation)*
- Polymorfismus (polymorphism)
- Dědičnost (inheritance)

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

- třídě založené jazyky (Java) \Rightarrow dědičnost tříd

Přepisování (overriding)

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

Přetěžování (overloading)

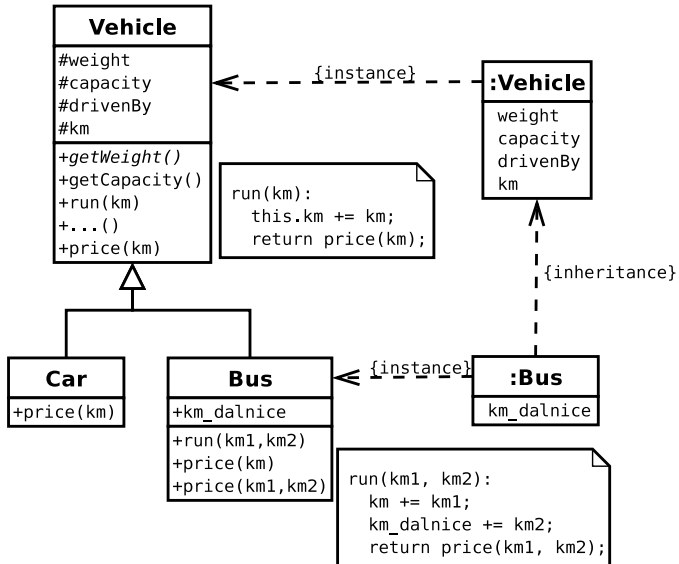
- technika vícenásobné definice operace v jedné třídě.
- Java:

```
prevedNa(Ucet u, int castka);  
prevedNa(Ucet u);
```

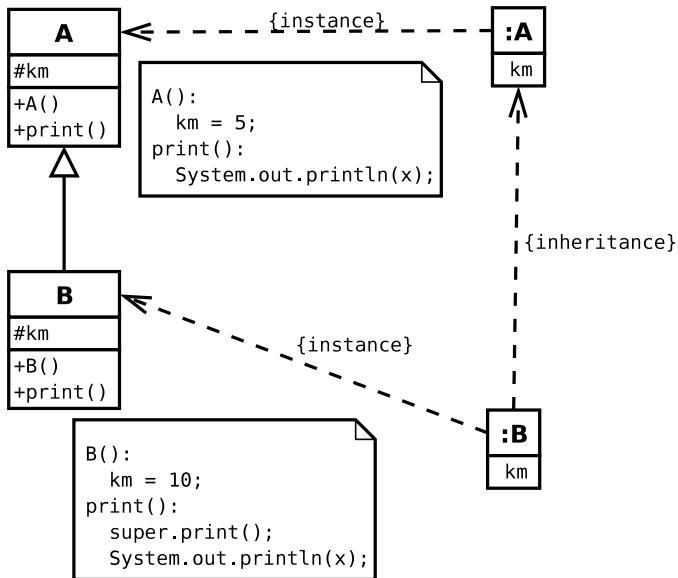
- Smalltalk nezná přetěžování:

```
preved: castka na: u.  
prevedNa: u.
```

Třídně orientované jazyky – dědičnost

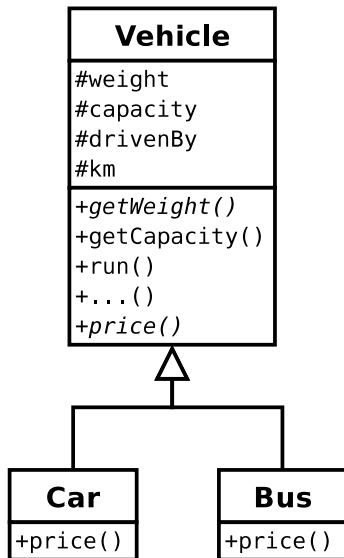


Třídně orientované jazyky – dědičnost



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!!!*



```
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 ...  
    }  
}
```

```
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;
    }
    ...
}
```

Základní kroky

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

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

```
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);
    }
    ...
}
```

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
            ...
    }
}
```

Skládání

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

Po vytvoření a naplnění instance třídy \mathbf{U}

- vytvoříme instanci třídy \mathbf{RU} a vložíme do ní inicializovanou instanci třídy \mathbf{U}

Dědičnost versus skládání

```
class U { ... }

class RU {
    protected U inner;

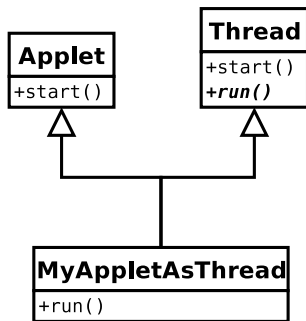
    public RU(U u) {
        inner = u;
    }

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

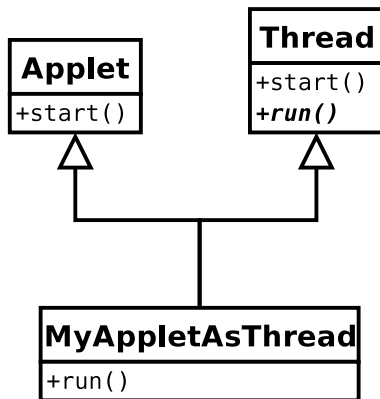
Vícenásobná dědičnost

Vícenásobná dědičnost

- komplikuje návrh (čitelnost)
- problém nejednoznačnosti
- dá se obejít (skládání objektů)
- existují případy, kdy má vícenásobná dědičnost význam

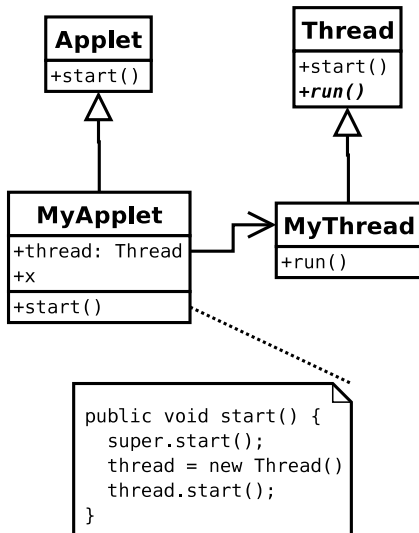


Java nemá vícenásobnou dědičnost!

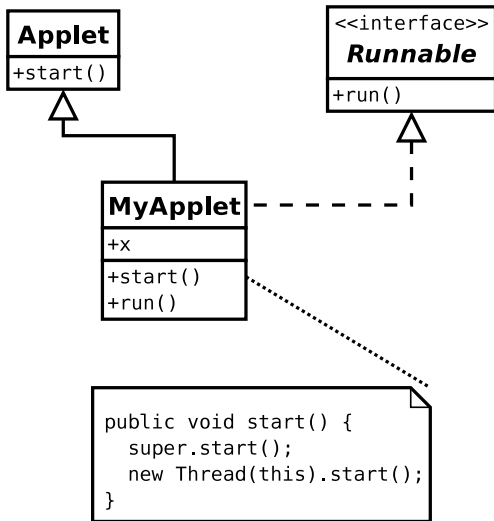


Vícenásobná dědičnost

Jak se obejít bez vícenásobné dědičnosti – I



Jak se obejít bez vícenásobné dědičnosti – II



Rozhraní objektu

- množina operací, které objekt nabízí
- pouze definuje co objekt umí (nabízí), *ne*definuje *jak*
- způsob provedení operace závisí na implementaci (metoda)
 - stejné rozhraní může být implementováno různými objekty
 - stejné operace mohou mít různé implementace

Polymorfismus

- mnohotvarost, schopnost výskytu v mnoha formách
- výskyt různých typů chování na základě stejné zprávy
 - možnost vícenásobné definice operace s jedním názvem, která tak může nabývat více implementací (implementuje různé chování).
 - logický vztah podobných operací (aplikace operací na podobné, ale technicky různé situace)

```
obj.start();
```

Rozhraní

- specifikuje množinu vlastností, ale *neimplementuje je*
- vlastnostmi se myslí především operace
- definuje *typ* objektu

Třída (také poněkud nepřesně zvaná objektový typ)

- implementuje rozhraní (tj. všechny metody rozhraní)
- třída představuje vzor pro vytváření objektů
- třída představuje skupinu objektů, které nesou stejné vlastnosti (kvalitativně)
 - např. všechny objekty třídy `Clовек` mají vlastnost `jmeno`,
 - tato vlastnost má však obecně pro různé lidi různé hodnoty – lidi mají různá jména
- *pozn.: třída sama o sobě deklaruje rozhraní ⇒ třída také definuje typ objektu*

Objekt

- objekt je jeden konkrétní jedinec (reprezentant, entita) příslušné třídy
- pro konkrétní objekt nabývají vlastnosti deklarované třídou konkrétních hodnot
- objekt je *instancí* třídy