

Seminář Java

Programování v Javě – II

Radek Kočí

Fakulta informačních technologií VUT

Únor 2008

- Rozhraní: použití, dědičnost
- Hierarchie dědičnosti
 - typová konverze
 - typová inference
- Porovnávání objektů

Význam typování

- určit sémantický význam elementům (hodnoty v paměti)
- pokud má paměťová hodnota přiřazený typ, můžeme s ní pracovat na vyšší úrovni – víme jaké operace je možné provést, můžeme provádět kontrolu typové konzistence atp.

Statically typované jazyky

- k typové kontrole dochází v době kompilace
- *C++*, *Java*, ...

Dynamicky typované jazyky

- k typové kontrole dochází v době běhu programu
- *Smalltalk*, *Self*, *Python*, *Lisp* ...

```
| x |  
x := self get.  
x message.           "překlad:  neznámý typ"
```

```
Type x;  
x = this.get();  
x.message();    //překlad:  Type a odvozené
```

Dynamická kontrola

- probíhá u všech jazyků
- jako dynamicky typované se označují ty, které nemají statickou kontrolu
- některé staticky typované jazyky (*C + +*, *Java*) umožňují dynamické přetypování, čímž částečně obcházejí statickou typovou kontrolu

- 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 podtříd třídy `A`.

- Výsledkem je vždy řetězec, ale argumenty mohou být i jiných typů, např.

- např.

```
System.out.println("objekt o = " + 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)

Operátory typové konverze (přetypování)

- píše se (`typ`) hodnota
- např. (`int`) `o`, kde `o` byla proměnná deklarovaná jako `long`.
- s konverzí primitivního typu může dojít ke změně hodnoty
- 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 prostřednictvím operátoru `==` (`!=`)

- **true** \Rightarrow jedná se o dva odkazy na tentýž objekt – tj. o dva totožné objekty
- **false** \Rightarrow 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 rovnocennosti**

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

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;  
    }  
}
```

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();
    }
}
```

V Javě, na rozdíl od C++ neexistuje vícenásobná dědičnost

- to nám ušetří řadu komplikací (problém nejednoznačnosti)
- ale je třeba to něčím nahradit

Pokud po třídě chceme, aby disponovala vlastnostmi z několika různých množin (skupin), můžeme ji deklarovat tak, že

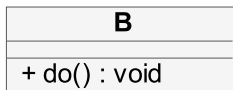
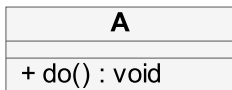
- implementuje více rozhraní

Rozhraní a třída

- rozhraní specifikuje množinu vlastností, ale *neimplementuje je*
- rozhraní definuje *typ* objektu
- objekt je typu A, pokud její třída implementuje rozhraní A
- *třída sama o sobě deklaruje rozhraní* \Rightarrow *třída také definuje typ objektu*
- objekt může mít více typů (implementovat více rozhraní)

Typová zaměnitelnost

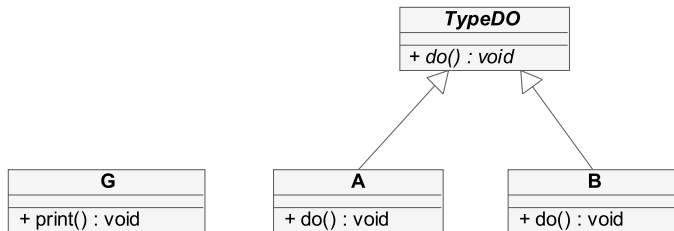
- Do proměnné, jejíž typ je deklarován jako třída **A**, lze dosadit všechny instance třídy **A** a všechny instance podtříd třídy **A**.
- Do proměnné, jejíž typ je deklarován jako rozhraní **A**, lze dosadit všechny instance tříd, které implementují rozhraní **A**.



```
public void m1(A obj) { obj.do(); }
```

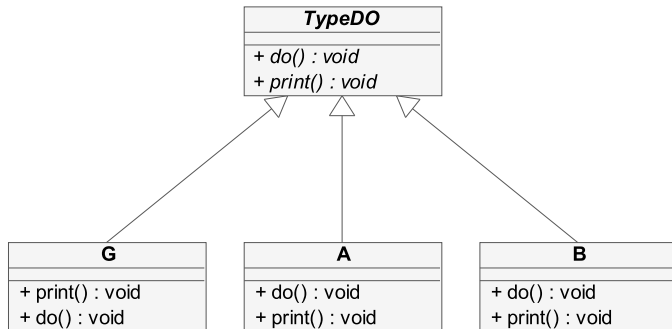
```
m1(new A());
```

```
m1(new B()); <- !
```



```
public void m1(TypeDO obj) { obj.do(); }  
m1(new A());  
m1(new B());
```

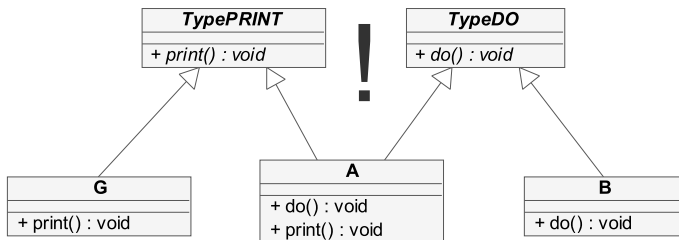
```
public void m2(A obj) { obj.print(); }  
m2(new A());  
m2(new G()); <-!
```



```
public void m1(TypeDO obj) { obj.do(); }
public void m2(TypeDO obj) { obj.print(); }
```

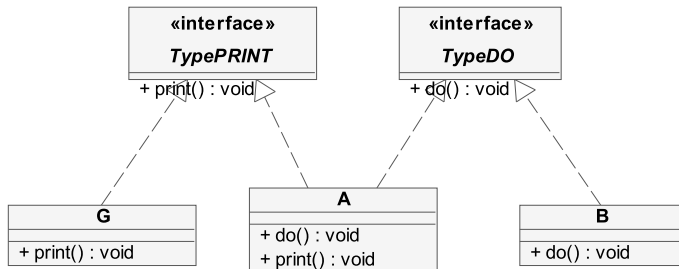
```
m1(new A()); <-- B, G
```

```
m2(new A()); <-- B, G
```



```
public void m1(TypeDO obj) { obj.do(); }
public void m2(TypePRINT obj) { obj.do(); }
```

```
method(new A()); <-- B
method(new A()); <-- G
```



```
public void m1(TypeDO obj) { obj.do(); }
public void m2(TypePRINT obj) { obj.do(); }
```

```
method(new A()); <-- B
method(new A()); <-- G
```

Co je rozhraní

- popis (specifikace) množiny vlastností (metod), aniž bychom tyto vlastnosti ihned implementovali.
- určitá třída implementuje rozhraní, pokud implementuje všechny metody, které jsou daným rozhraním předepsány.

Rozhraní v Javě je specifikováno

- množinou hlaviček metod označenou identifikátorem – názvem rozhraní
- ucelenou specifikací – tj. popisem, co přesně má metoda dělat (vstupy/výstupy metody, její vedlejší efekty ...)

- Vypadá i umísťuje se do souborů podobně jako deklaráce třídy
- Všechny metody v rozhraní musí být `public` a v hlavičce se to ani nemusí uvádět.
- Všechny metody v rozhraní jsou zároveň automaticky abstraktní \Rightarrow těla metod se neuvádějí.
- Rozhraní může obsahovat proměnné – jedná se vždy o konstantu (modifikátor `final` se uvádět nemusí)

Příklad deklaráce rozhraní

```
public interface Informator {  
    public void vypisInfo();  
}
```


Implementace rozhraní

```
public class Ucet implements Informator {  
    ...  
    public void vypisInfo() {  
        ...  
    }  
}
```

- Třída implementuje všechny metody předepsané rozhráním.
- Třída může implementovat více rozhraní současně.

```
public class Name implements Interfacel,  
                             Interface2  
{ ... }
```

- Tam, kde stačí funkcionalita definovaná rozhráním.
- Proměnnou můžeme definovat jako typ rozhraní (ne třídu, která rozhraní implementuje).
- Do proměnné lze přiřadit libovolný objekt, který **implementuje** uvedené rozhraní.
- Lze používat pouze metody deklarované rozhráním! (*viz dále ...*)
- Umožňuje větší flexibilitu kódu při zachování (statické) typové kontroly.

```
Informator petruvUcet = new Ucet("Petr");  
petruvUcet.vypisInfo();
```

- Podobně jako u tříd i rozhraní může být *děděno*.
- Třída dědí maximálně z jednoho předka.
- Rozhraní může dědit z více předků (*vícenásobná dědičnost*).

```
public interface DobryInformator
           extends Informator
{
    public void vypisViceInfo();
}
```

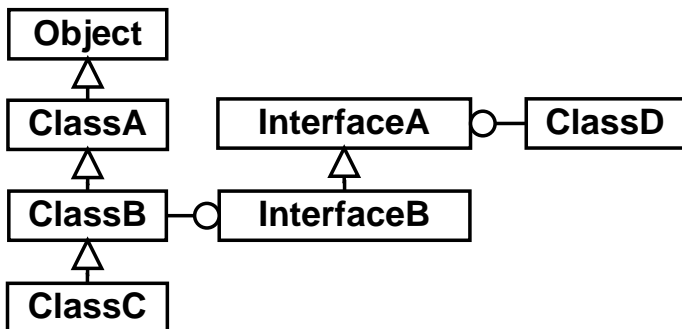
Třída, která implementuje rozhraní *DobryInformator* musí implementovat *obě* metody:

```
public class Ucet implements DobryInformator {
    ...
    public void vypisInfo() {
        ...
    }
    public void vypisViceInfo() {
        ...
    }
}
```

Typová zaměnitelnost

- Do proměnné, jejíž typ je deklarován jako třída **A**, lze dosadit všechny instance třídy **A** a všechny instance podtřídy třídy **A**.
- Do proměnné, jejíž typ je deklarován jako rozhraní **A**, lze dosadit všechny instance tříd, které implementují rozhraní **A**, nebo které jsou jejich podtřídami.
- Do proměnné, jejíž typ je deklarován jako rozhraní **A**, lze dosadit všechny instance tříd (a podtříd), které implementují rozhraní **A** nebo odvozené rozhraní.

Dosazení objektu do proměnné – I

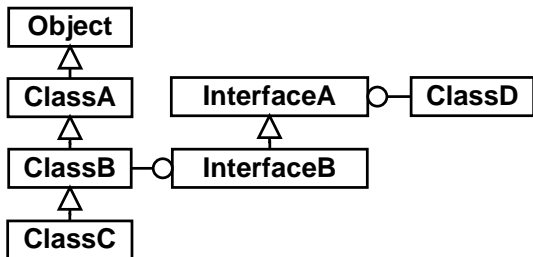


```
void method(ClassA o) { ... }
```

● ○ ⇒ **ClassA**, **ClassB**, **ClassC**

● ○ == **ClassB** ⇒ **(ClassB)** ○

Dosazení objektu do proměnné – II



```
void method(InterfaceB o) { ... }
```

- **o** ⇒ **ClassB, ClassC**

```
void method(InterfaceA o) { ... }
```

- **o** ⇒ **ClassB, ClassC, ClassD**

- **o == ClassC** ⇒ **(InterfaceB) o**

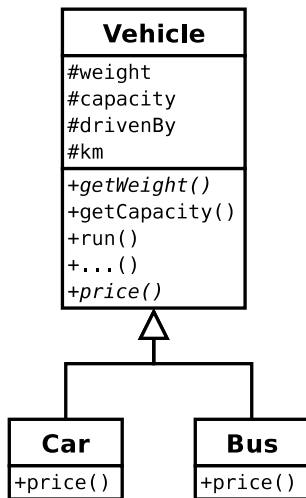
- **o == ClassC** ⇒ **(ClassC) o**

Dosazení objektu do proměnné – příklad

```
public abstract class Vehicle {
    public abstract int price(int km);
}
public class Car extends Vehicle {
    public int price(int km) { ... }
}
public class Bus extends Vehicle {
    public int price(int km) { ... }
}
```

```
method(new Bus());
public void method(Vehicle v) {
    Car c = (Car) v;    ← (ClassCastException)
    System.out.println(a.price(200));
}
```


Dosazení objektu do proměnné – příklad



Dosazení objektu do proměnné – příklad

```
(1) Car c = new Car();
```

```
(2) Bus b = new Bus();
```

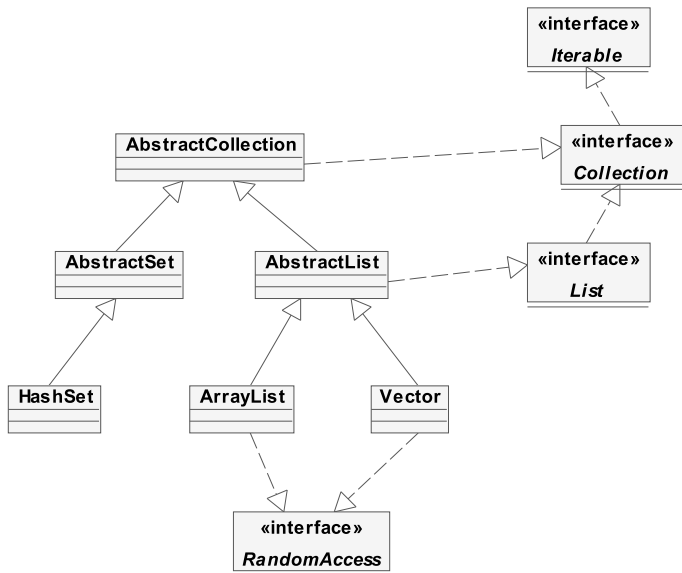
```
(3) Vehicle vc = c;
```

```
(4) Object ob = b;
```

```
(5) Car cc = (Car) ob;
```

```
(6) Bus bb = (Bus) ob;
```

```
(7) Car ccc = (Car) vc;
```



Rozhraní (**java.util**):

```
public interface Collection ...
```

Implementující třídy:

```
AbstractCollection, ArrayList, AbstractList, AbstractSet,  
ArrayList, BeanContextServicesSupport,  
BeanContextSupport, HashSet, LinkedHashSet,  
LinkedList, TreeSet, Vector
```

Třída **Vector**:

```
public class Vector ... {  
    ...  
    public Vector(Collection c) ...  
    ...  
}
```