Seminář Java

IV

Rekapitulace

- Deklarace tříd
  - Proměnné, metody, konstruktory, modifikátory přístupu
- Datové typy
  - primitivní, objektové, pole
- Dědičnost
- Řídící konstrukce
  - Podmínky, cykly

Abstraktní třídy

- Třída, která danou specifikaci implementuje jen *částečně*.
  - *Abstraktní třída* = částečná implementace
  - *Třída* = úplná implementace
- Abstraktní třída *nemůže* mít instance.

```
public abstract class GraphicObject {
    int x, y;
    . . .
    void moveTo(int newX, int newY) {
        . . .
    }
    abstract void draw();
}
```

Abstraktní třídy

```
class Circle extends GraphicObject {
    void draw() {
        . . .
    }
}

class Rectangle extends GraphicObject {
    void draw() {
        . . .
    }
}
```

Modifikátor `final`

- Deklaruje konečný (neměnný) stav
- Třídy
  - `public final class Ucet { ... }`
  - od této třídy nelze "dědit" (vytvářet její potomky)
- Metody
  - `public final void print() { ... }`
  - tato metoda nemůže být "překryta" (overloaded) v odvozených třídách (potomci)
- Proměnné
  - `protected final int i = 10;`
  - `protected final String s = "řetězec";`
  - `protected final Banka b = new Banka();`
  - obsah proměnné je neměnný
  - konstanta

Rozhraní

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

Co je rozhraní

- Rozhraní je vlastně popis (specifikace) množiny vlastností, aniž bychom tyto vlastnosti ihned implementovali.
- Vlastnostmi zde rozumíme především metody.
- Říkáme, že určitá třída implementuje rozhraní, pokud implementuje všechny metody, které jsou daným rozhraním předepsány.
- Javové rozhraní je tedy
  - množina hlaviček metod označená identifikátorem – názvem rozhraní
  - + ucelené specifikace – tj. popis, co přesně má metoda dělat (vstupy/výstupy metody, její vedlejší efekty . . .)

Deklarace rozhraní

- Vypadá i umisťuje se do souborů podobně jako deklarace 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 deklarace rozhraní

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

Implementace rozhraní

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

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

```
public class Name implements Interface1, Interface2
{ ...  }
```

Použití rozhraní

- Tam, kde stačí funkcionalita definovaná rozhraním.
- Proměnnou můžeme definovat jako typ rozhraní (ne třídu, která rozhraní implementuje).
- Do proměnné lze přiřadit libovolný objekt, který implementuje uvedené rozhraní.

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

- Deklarace, že třída implementuje rozhraní ji nezavazuje, poskytuje typovou informaci o třídě.
- Lze používat pouze metody deklarované rozhraním! *(viz dále . . .)*
- Umožňuje větší flexibilitu kódu při zachování (statické) typové kontroly.

Použití rozhraní

Rozhraní (`java.util`):

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

Implementující třídy:

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

Třída `Vector`:

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

Rozšiřování rozhraní

- 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();
}
```

Rozšiřování rozhraní

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

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

Rekapitulace

- Známe
  - *Rozhraní* = specifikace
  - *Abstraktní třída* = částečná implementace
  - *Třída* = úplná implementace
- Umíme deklarovat třídu a její vlastnosti.
- Umíme vytvářet instance tříd a volat její metody.
- Umíme vytvářet specializované třídy a rozhraní.
- Umíme implementovat rozhraní.

Operátory a výrazy, porovnávání objektů

- Aritmetické
- Logické
- Relační
- Bitové
- Operátor podmíněného výrazu ? :
- Operátory typové konverze (přetypování)
- Operátor zřetězení +
- Porovnávání objektů
- Priority operátorů a vytváření výrazů

Aritmetické operátory

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

Operátor podmíněného výrazu ? :

Bitové:

- 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);
```

Operátor zřetězení +

- Výsledkem je vždy řetězec, ale argumenty mohou být i jiných typů, např.
- následující sekvence je v pořádku

```
int i = 1; System.out.println("promenna i=" + i);
```

- s řetězcovou konstantou se spojí řetězcová podoba dalších argumentů (např. čísla).
- Pokud je argumentem zřetězení odkaz na objekt 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)

Relační (porovnávací) operátory

- Tyto lze použít na porovnávání primitivních hodnot:
  - `<`, `<=`, `>=`, `>`
- Test na rovnost/nerovnost lze použít na porovnávání primitivních hodnot i objektů:
  - `==`, `!=`
  - pozor na porovnávání objektů: `==` vrací true jen při rovnosti odkazů, tj. jsou-li objekty identické. Rovnost obsahu (tedy "rovnocennost") viz dále ...
  - 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`

Operátory typové konverze (přetypování)

- Podobně jako v C/C++
- Píše se `( typ )` hodnota
- 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`.
- Naproti tomu u primitivních typů se jedná o úpravu hodnoty – např. `int` přetypujeme na `short` a "ořeže" se tím rozsah.

Hierarchie dědičnosti

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

Porovnávání objektů

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 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

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;
    }
    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;
    }
}
```

Metoda `hashCode`

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

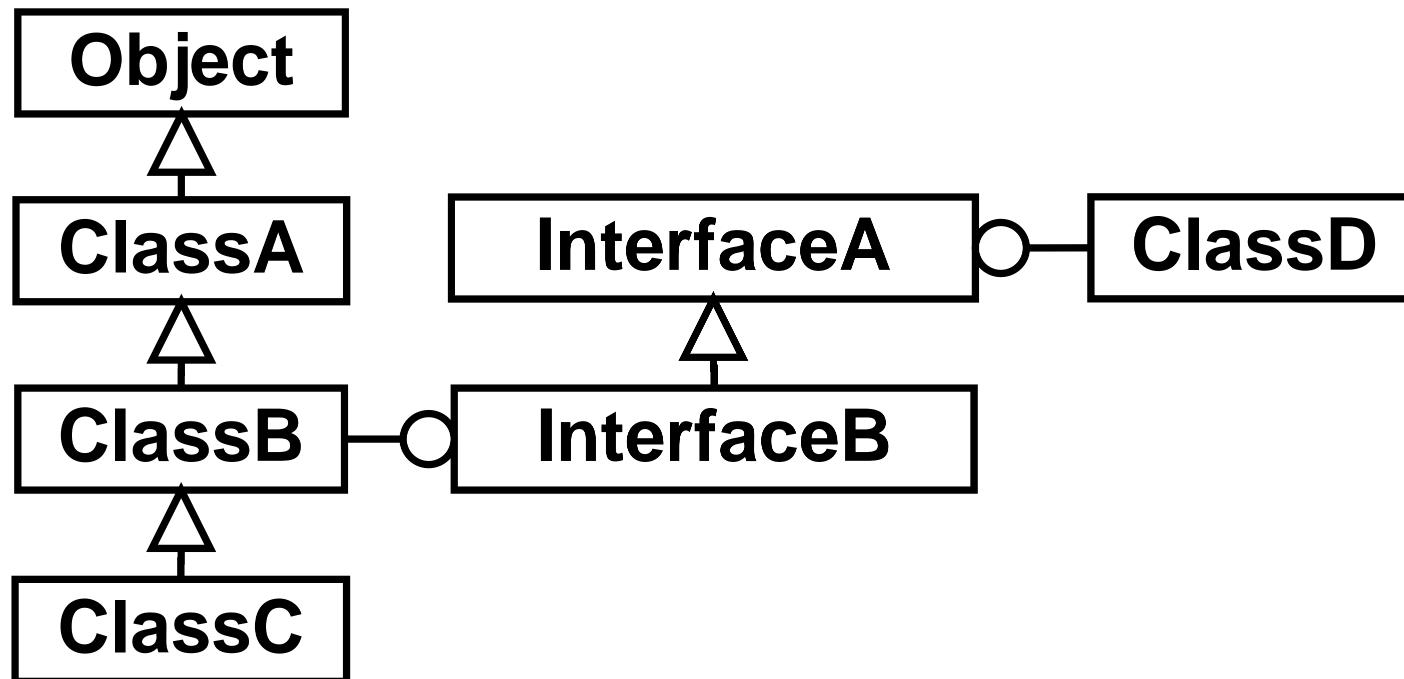
- `hashCode` vrací celé číslo (`int`) "co nejlépe" charakterizující obsah objektu
- pro dva stejné (`equals`) objekty musí *vždy vrátit stejnou hodnotu*
- pro dva obsahově různé objekty by `hashCode` naopak měl vracet různé hodnoty (ale není to stoprocentně nezbytné a ani nemůže být vždy splněno)

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();
    }
}
```

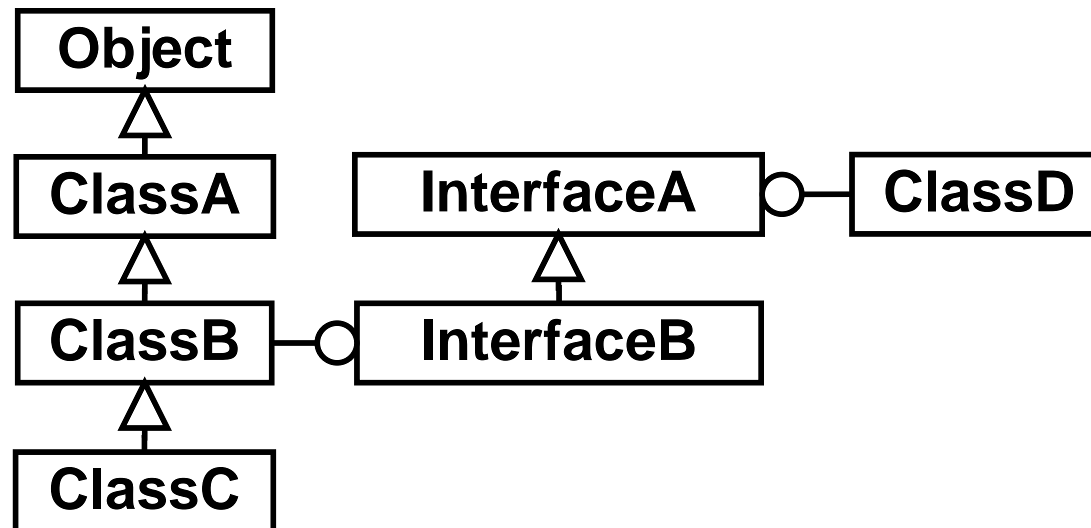
Dosazení objektu do proměnné – I




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

- o ⇒ `ClassA`, `ClassB`, `ClassC`
- `o == ClassB` ⇒ `(ClassB) o`

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

```
class A {
    int i = 10;
    public int value() { return i; }
}

class B {
    int i = 20;
    public int value() { return i; }
}
```

```
    public void method1() {
        ...
        method2(new B());
    }
    public void method2(Object o) {
        A a = (A) o;    ⇐ chyba (ClassCastException)
        System.out.println(a.value());
    }
```

Top-level classes

Třídy nejvyšší úrovně (top-level classes)

- "normální třídy" – jsou přímými členy nějakého balíku

```
public class TopLevel1 {
    ...
}
```

Vnořené třídy (inner classes)

Lokální třídy

- vnořené v jiné třídě (na úrovni lokálních proměnných)
- uváděné uvnitř bloku (platné pouze v uvedeném bloku)
- nesmí být *public*, *private* a *protected*

```
public class TopLevel1 {
    private String text = "interni promenna";
    public void test() {
        {
            class A {
                public A() { System.out.println(text); }
            }
            A a = new A();
        }
        // tady už není třída A dostupná
    }
}
```

Vnořené třídy (inner classes)

Členské třídy (member classes)

- vnořené v jiné třídě (na úrovni vlastností třídy)

```
public class TopLevel1 {
    private String text = "interni promenna";
    class Inner {
        public Inner() {
            System.out.println("Trida Inner: " + text);
        }
    }
}
```

Vnořené třídy (inner classes)

Vnořené top-level třídy

- členské třídy s modifikátorem `static`
- vnořená rozhraní
- mohou být *public*, *private* a *protected*

```
public class TopLevel1 {
    private String text = "interni promenna";
    static public class TopLevel2 {
        public TopLevel2() {
            TopLevel1 t = new TopLevel1();
            System.out.println(t.text);
        }
    }
    interface Cool {
        ...
    }
}
```

Vnořené třídy (inner classes)

Vnořené top-level třídy

- modifikátor `static` má jinou sémantiku než u vlastností tříd!
- používá se k seskupení souvisejících tříd bez nutnosti vytvářet nový balík
- přístup k vnořeným top-level třídám (rozhraním)

```
new TopLevel1.TopLevel2();
```

Vnořené třídy

Anonymní třídy (anonymous classes)

- zvláštní případ vnořené třídy

```
new Typ ( parametry ) {
    tělo anonymní třídy
}
```

- Typ představuje
  - jméno konstruktoru rodičovské(!) třídy, od které je anonymní třída odvozena (následují jeho parametry)
  - jméno rozhraní – anonymní třída jako jediná může přímo instanciovat rozhraní (zde se parametry neuvádějí)

```
class NejakaTrida {
   Runnable r = new Runnable() {
      public void run() {
         // ...
      }
   }
}
```

Souhrn

- Deklarace tříd
  - Proměnné, metody, konstruktory, modifikátory přístupu
- Datové typy
  - primitivní, objektové, pole
- Řídící konstrukce
  - Podmínky, cykly
- Dědičnost
- Rozhraní
- Porovnávání objektů
- Testování
- Tvorba dokumentace