Seminář Java
IV
2005/2006

Radek Kočí

Rekapitulace

- Třídy: proměnné, metody, konstruktory, modifikátory přístupu, dědičnost
- Datové typy: primitivní, objektové
- Řídící konstrukce, operátory
- Ladění

Téma přednášky

- Abstraktní třídy
- Rozhraní: použití, dědičnost
- Hierarchie dědičnosti: typová konverze, typová inference
- Porovnávání objektů
- Pole
- Vnořené třídy

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

Rozhraní

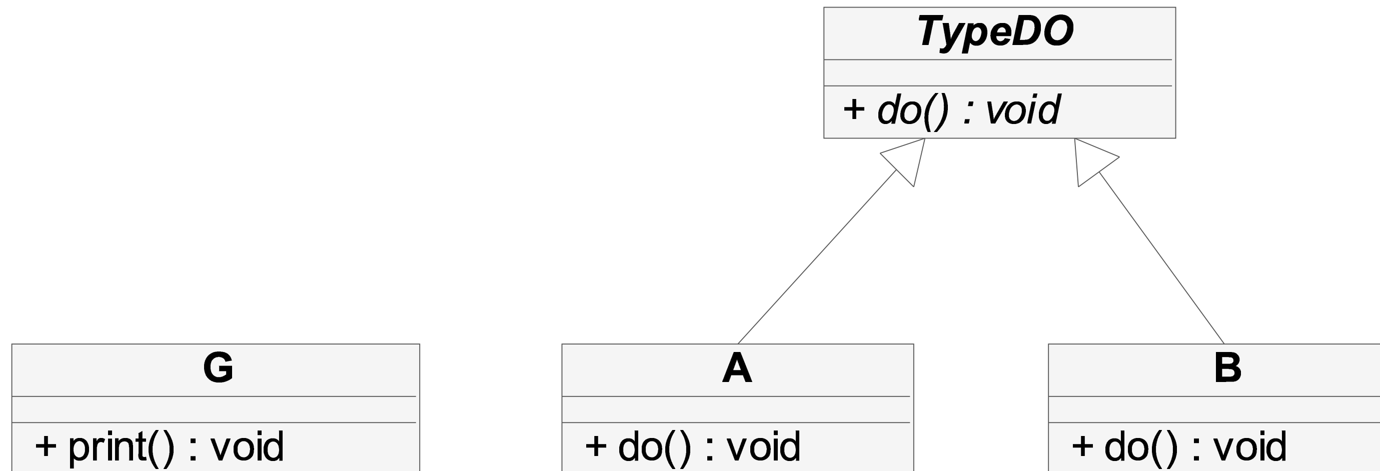
A
+ do() : void

B
+ do() : void

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

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

Rozhraní



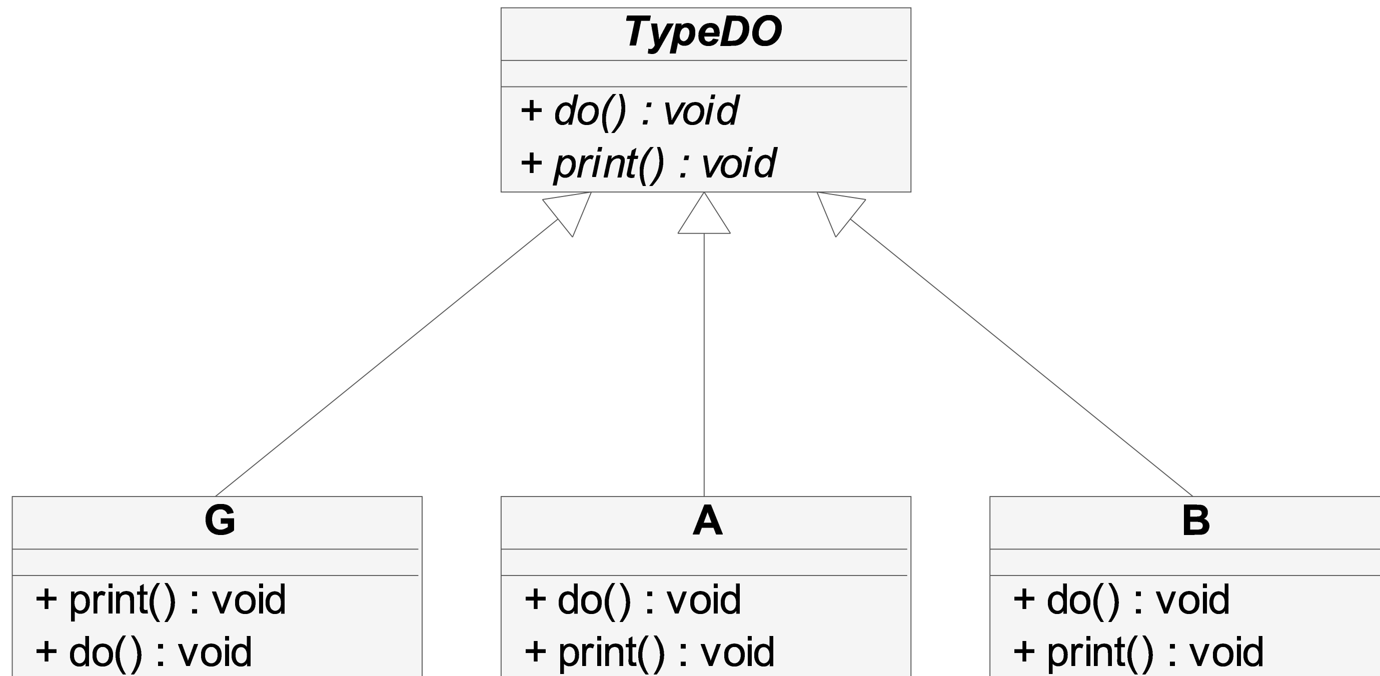

```
public void m1(TypeDO obj) { obj.do(); }

m1(new A());
m1(new B());

public void m2(A obj) { obj.print(); }

m2(new A());
m2(new G()); <-!
```

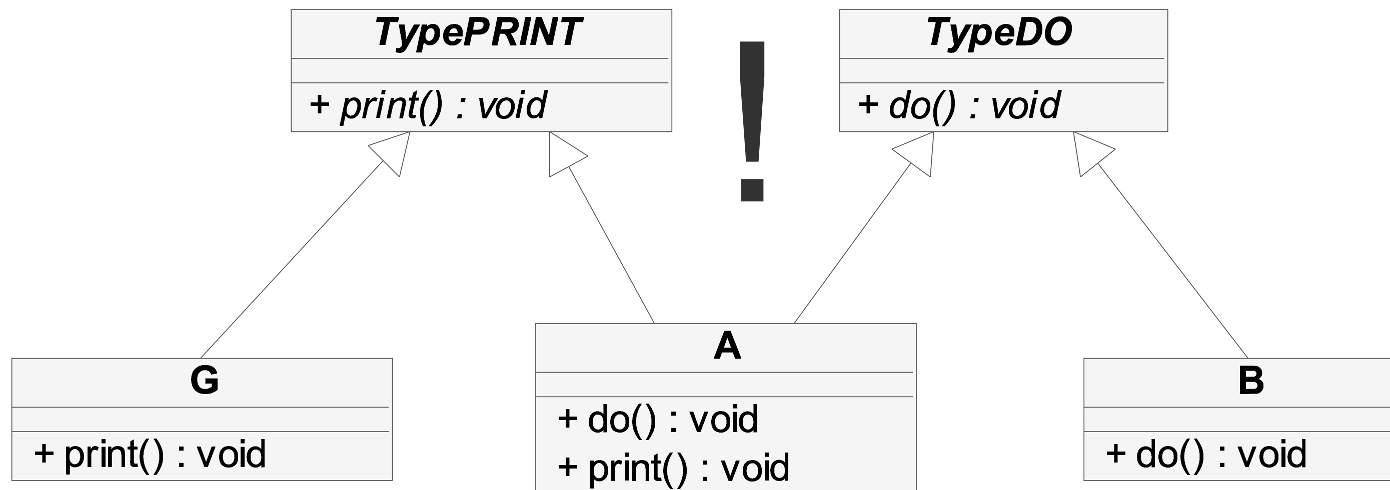
Rozhraní




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

m1(new A()); <- B, G
m2(new A()); <- B, G
```

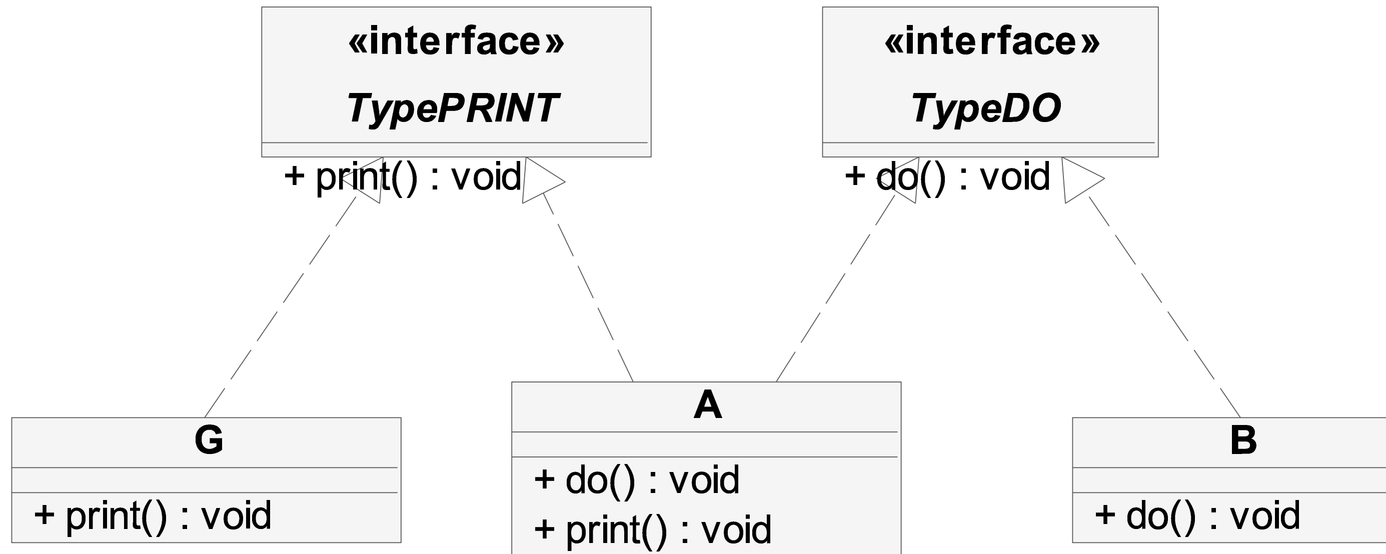
Rozhraní




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

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

Rozhraní




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

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

Rozhraní

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 . . .)

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 ⇒ 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.

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*).

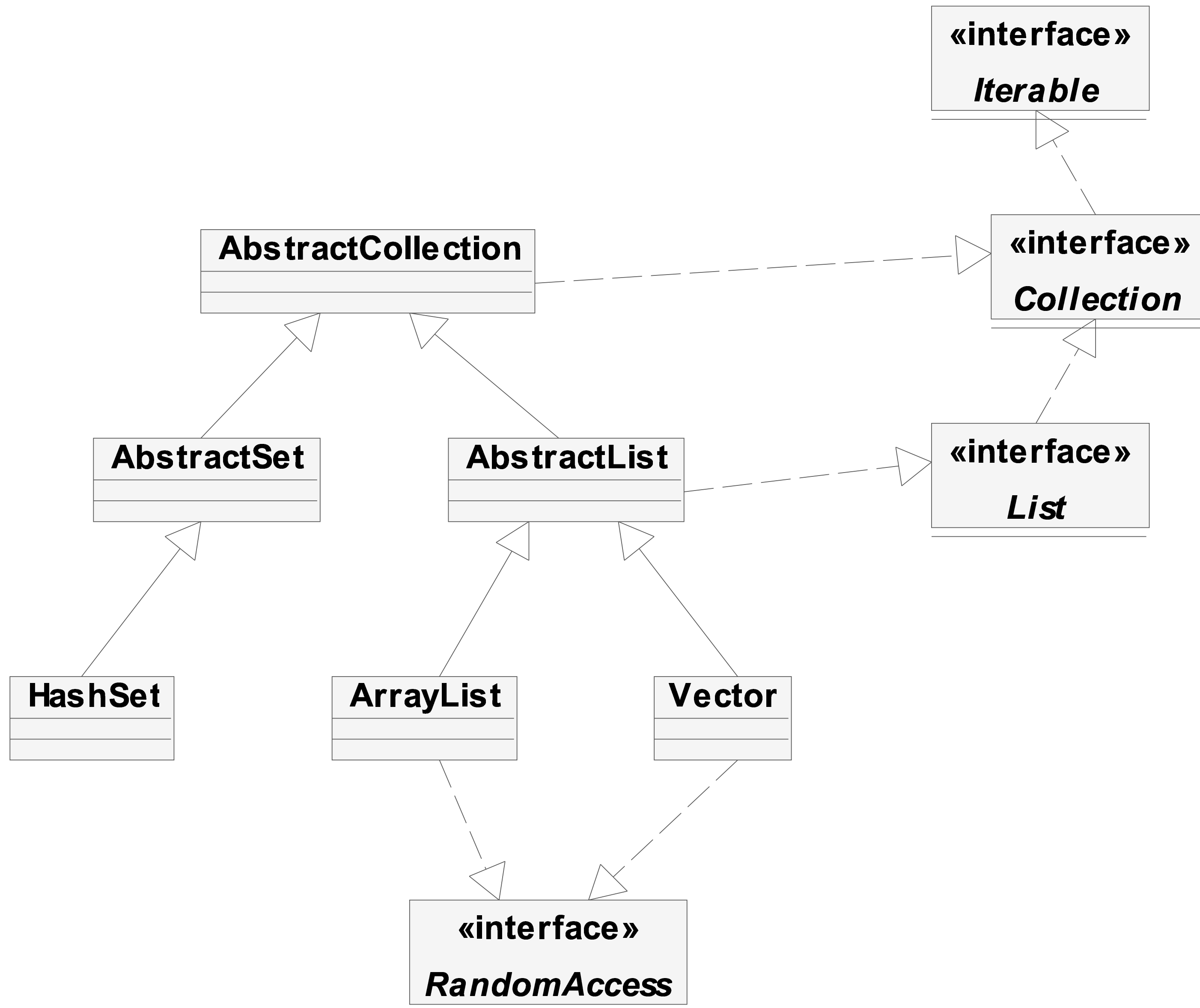
```
public interface DobryInformator extends Informator {
     public void vypisViceInfo();
}
```

Rozšiřování rozhraní

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

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

Použití rozhraní



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) ...
     ...
}
```

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.

Hierarchie dědičnosti

- 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 podříd třídy `A`.

Operátor zřetězení +

- 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. `(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

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()`:

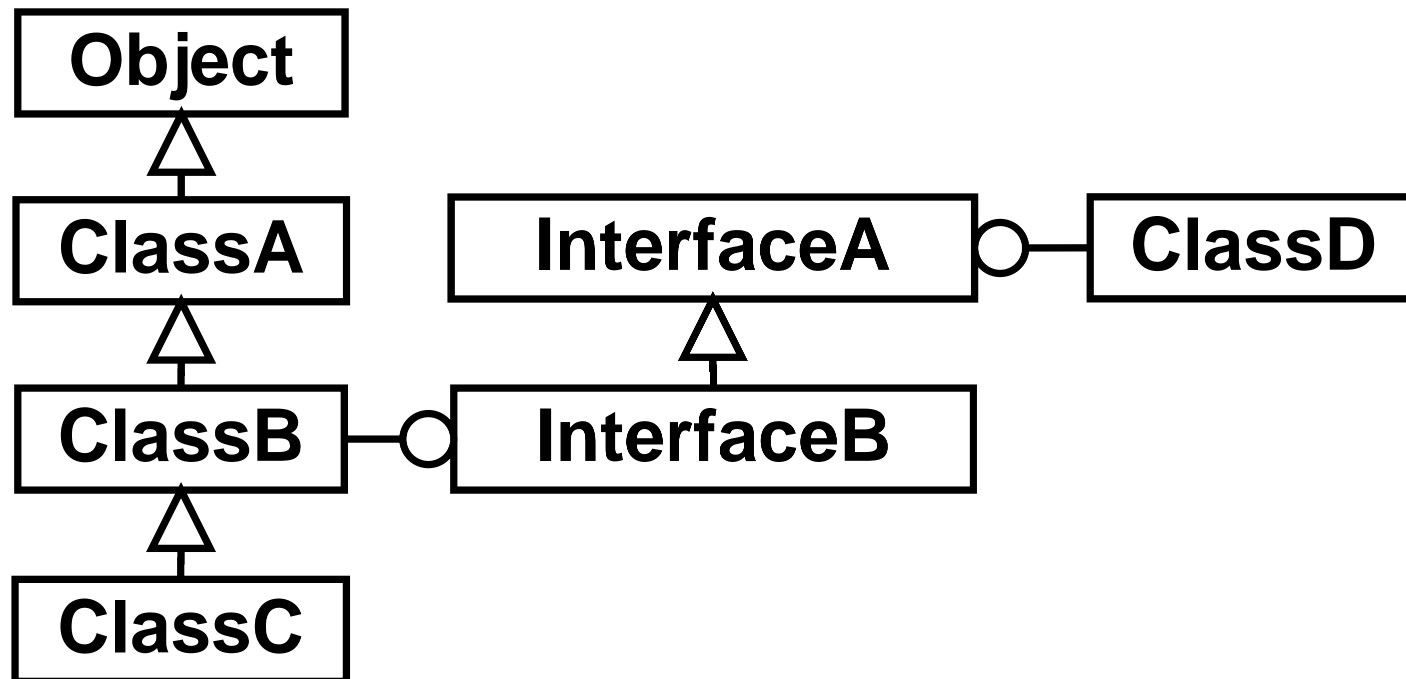
- `hashCode` vrací celé číslo (`int`) "co nejlépe" charakterizující obsah objektu
- pro dva stejné (`equals`) objekty musí *vždy vrátit stejnou hodnotu*
- pro dva obsahově různé objekty by `hashCode` naopak měl vracet různé hodnoty (ale není to stoprocentně nezbytné a ani nemůže být vždy splněno)

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();
    }
}
```

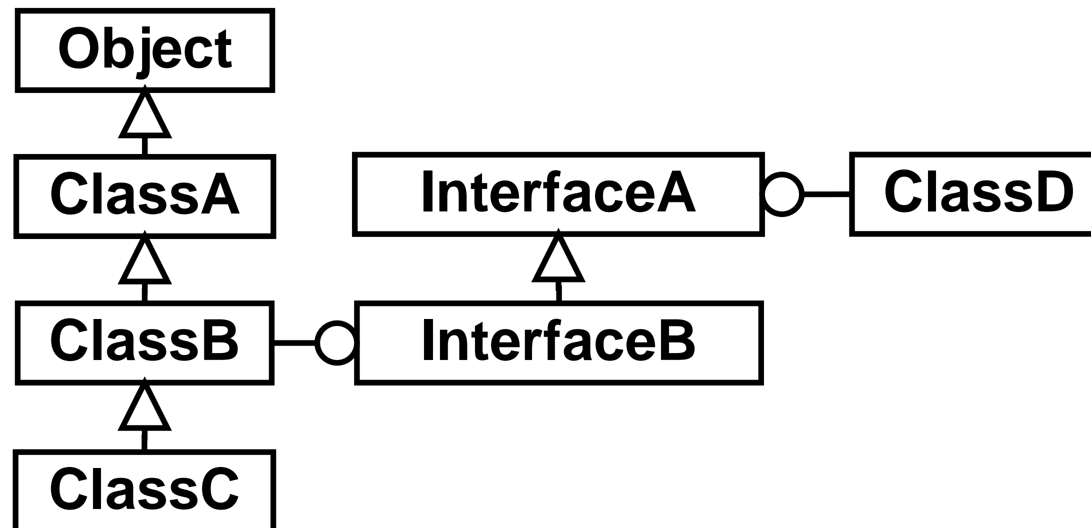
Dosazení objektu do proměnné – I




```
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() {
         // ...
      }
   }
}
```

Pole

- Pole v Javě je speciálním objektem.
- Můžeme mít pole jak primitivních, tak objektových hodnot
  - pole primitivních hodnot tyto hodnoty obsahuje
  - pole objektů obsahuje odkazy na objekty
- Kromě pole v Javě existují i jiné objekty na ukládání více hodnot – *kontejnery* (bude později ...)

Pole – I

Před použitím je nutné pole

- deklarovat
- vytvořit
- inicializovat (naplnit)

Syntaxe deklarace

- `typhodnoty [] identifikator`
- na rozdíl od C/C++ nikdy neuvádíme při deklaraci počet prvků pole – ten je podstatný až při vytvoření objektu pole

Pole – II

Vytvoření pole

- jako u jiného objektu – voláním konstruktoru:
  - `nazevPole = new typhodnoty [počet_prvků];`
- nebo inicializací při deklaraci:
  - `int [] nazevPole = { 1, 2, 3};`

Syntaxe přístupu k prvkům

- `identifikator [ indexprvku ]`
- přiřazení prvku do pole:
  `identifikator [ indexprvku ] = hodnota;`
- čtení hodnoty z pole:
  `proměnná = identifikator [ indexprvku ];`

Pole – III

```
Ucet [] ucty;                        // deklarace pole
ucty = new Ucet[5];                  // vytvoření pole
ucty[0] = new Ucet("Franta");   // vytvoření objektu
                // a inicializace 1. prvku pole !!!

ucty[0].vypisInfo();                // přístup k prvku pole
```

- V poli `ucty` je naplněn 1. prvek odkazem na objekt
- Ostatní prvky zůstaly naplněny prázdnými odkazy `null`.

Pole – IV

Co když vynecháme vytvoření pole?

```
Ucet [] ucty;
ucty[0] = new Ucet("Franta");  // chyba, pole neexistuje
```

Co když vynecháme inicializaci pole?

```
Ucet [] ucty;
ucty = new Ucet[5];
ucty[0].vypisInfo();          // chyba, prvek neexistuje
```

Kopírování pole

Přiřazení proměnné objektového typu (a tedy i polí) vede pouze k duplikaci odkazu, nikoli celého odkazovaného objektu.

```
Ucet [] ucty = new Ucet[5];
Ucet [] ucty2;
ucty2 = ucty;
```

- Proměnná `ucty2` obsahuje odkaz na stejné pole jako `ucty`.

```
Ucet [] ucty2 = new Ucet[5];
System.arraycopy(ucty, 0, ucty2, 0, lidi.length);
```

- Proměnná `ucty2` obsahuje kopii původního pole.
- Také `arraycopy` však do cílového pole zduplikuje jen odkazy na objekty, nevytvoří kopie objektů!

Výpis argumentů programu

```
public class Pole {
    public static void main(String[] args) {
        for(int i=0; i<args.length; i++)
            System.out.println(args[i]);
    }
}
```