Dědičnost

Tomáš Pitner, Radek Ošlejšek, Marek Šabo

Vtip

> Dědičnost. Nejlepší objektově-orientovaný způsob, jak zbohatnout.

Dědičnost

- Objektové třídy jsou obvykle **podtřídami**, tj. speciálními případy jiných tříd:

```
class Subclass extends Superclass
```

```
class DogKeeper extends Person {
// methods & attributes for DogKeeper
// in addition to Person
}
```

- Všechny objekty typu `DogKeeper` jsou současně typu `Person`.

```
Person p = new DogKeeper("Karel");
```

V Javě dědí **každá** třída od třídy `Object`.

Definice

- **Nadtřída** = superclass, "bezprostřední předek", "rodičovská třída"
- **Podtřída** = subclass, "bezprostřední potomek", "dceřinná třída"
  - je specializací své nadtřídy
  - přebírá vlastnosti nadtřídy
  - zpravidla přidává další vlastnosti, čímž nadtřídu *rozšiřuje* (`extends`)

Správné použití

- Dědičnost by měla splňovat vztah **je (is a)** — Chovatel psa *je* osoba.
- Při atributech třídy se používá vztah **má (has a)** — Osoba *má* jméno.

Proč používat dědičnost?

- Abychom zohlednili **konceptuální vztah** *obecnější* vs. *speciálnější* typ.
- Abychom se **vyhnuli opakování kódu** a dosáhli *znovupoužití* (= kód metod a atributů se podědí, nemusíme jej znovu psát).
- Mělo by platit oboje, aby mělo smysl dědičnost použít.

Tranzitivní dědění

Dědění může být vícegenerační:

```
public class Manager extends Employee { ... }

public class Employee extends Person { ... }
```

Manažer podědí metody a atributy ze třídy *Zaměstnanec* i *Osoba*.

Vícenásobná dědičnost

- Java vícenásobnou dědičnost u tříd **nepodporuje**!
- Důvodem je problém typu diamant:

```
class DoggyManager extends Employee, DogKeeper { }
class Employee  { public String toString() { "Employee";  } }
class DogKeeper { public String toString() { "DogKeeper"; } }

DoggyManager().toString(); // we have a problem!
```

- Vícenásobná dědičnost je možná jedině u rozhraní (metody zatím nemají definovanou implementaci).

Dědičnost a vlastnosti tříd

- Dědičnost (v kontextu Javy) znamená:
  1. potomek dědí **všechny** vlastnosti nadtřídy
  2. poděděné vlastnosti potomka se **mohou změnit** (např. překrytím metody)
  3. potomek může **přidat** další vlastnosti
- **Pozn.:** *Vlastnosti třídy* = metody & atributy třídy

Dědičnost vs. rozhraní

Dědičnost

- vyhýbáme se duplikaci kódu
- kód je kratší
- když potřebuji upravit předka, musím upravit změny ve všech potomcích, co může být netriviální

Použití rozhraní

- méně závislostí, více kódu
- více používné v praxi

Příklad s `Account`

- Vylepšíme třídu `Account` tak, aby zůstatek nebyl nižší než minimální zůstatek
- Realizujeme tedy změnu metody `debit` pomocí jejího *překrytí* (overriding)
- Stará třída `Account`:

```
public class Account implements Informing {
   private int balance;
   ...
   public boolean debit(int amount) {
      if(amount <= 0) return false;
      balance -= amount;
      return true;
   }
}
```

Nová třída `CheckedAccount`

```
public class CheckedAccount extends Account {
  private int minimalBalance;
  public CheckedAccount(Person owner, int minBal, int initBal) {
    super(owner, initBal); // calling Account constructor
    if(initBal < minBal) { // is initial balance sufficient?
       throw new IllegalArgumentException("low initial balance");
    }
    this.minimalBalance = minBal;
  }

  @Override
  public boolean debit(int amount) {
    // check min. balance
    if(getBalance() - amount >= minimalBalance) {
       return super.debit(amount); // the debit is deducted
    } else return false;
  }
}
```

Co tam bylo nového

- Klíčové slovo `extends` značí, že třída `CheckedAccount` je potomkem (*subclass*) třídy `Account`.

- Konstrukce `super.metoda(…);` značí, že je volána metoda předka, tedy třídy `Account`.
- Kdyby se `super` nepoužilo, zavolala by se metoda `debit` třídy `CheckedAccount` a program by se zacyklil!

V konstruktoru potomka je **vždy** nutno zavolat konstruktor předka, protože se v něm inicializují vlastnosti dané třídy.

Konstruktor předka je nutno zavolat jako **první**, protože by se pak dali použít vlastnosti, které zatím nejsou nainicializované.

Kompozice

Často se používá *skládání* (kompozice) objektů, kdy objekt **nedědí**, ale nese odkaz na jiný objekt.

```
public class CheckedAccount {

  private int minimalBalance;
  private Account account;

  public CheckedAccount(Person owner, int minBal, int initBal) {
    account = new Account(owner, initBal);
    ...
  }
  // account.debit(amount)
}
```

Kompozicí se zabývá navazující kurz *PV168 Seminář Javy*.