

# Dědičnost

## Table of Contents

Vtip .....	1
Dědičnost .....	1
Co to znamená? .....	2
Definice .....	2
Správné vs špatné použití .....	2
Proč používat dědičnost? .....	2
Tranzitivní dědění .....	3
Vícenásobná dědičnost .....	3
Dědičnost a vlastnosti tříd .....	3
Dědičnost vs. rozhraní .....	3
Pravidla pro konstruktory podtříd .....	4
Příklad s <code>Account</code> .....	4
Nová třída <code>CheckedAccount</code> .....	4
Volání kódu nadtřídy .....	5
Volání kódu nadtřídy .....	5
Repl.it demo k dědičnosti - bankovní účty .....	5
Pozor na překrývání atributů .....	5
Not what we wished to do .....	6
Namísto dědění lze použít skládání (kompozice) .....	6
Kompozice pořádně .....	6
Replit.com demo k dědičnosti - geometrické útvary .....	7

## Vtip

Dědičnost. Nejlepší objektově-orientovaný způsob jak zbohatnout.

## Dědičnost

- Dědičnost je charakteristická vlastnost objektových jazyků se třídami, jako jsou Java, Python, C# a další.
- U beztřídních (klasický JavaScript) se může řešit pomocí *prototypů*.
- Objektové třídy jsou obvykle **podtřídami**, tj. speciálními případy, jiných tříd:

```
class DogKeeper extends Person {  
  // methods & attributes for DogKeeper
```

```
// in addition to Person
}
```

## Co to znamená?

- Všechny objekty typu `DogKeeper` jsou současně typu `Person`.
- Místo jiné osoby lze použít `DogKeeper`.

```
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 nadtřidy
- přebírá vlastnosti nadtřidy
- zpravidla přidává další vlastnosti, čímž nadtřidu *rozšiřuje* (`extends`)

## Správné vs špatné použití

### Správně

Dědičnost by měla splňovat vztah **je** — každý `DogKeeper` *je* `Person`.

### Špatně

Každé oddělení má osobu vedoucího; je ale špatně dědit `Department extends Manager` protože neplatí `Department je Manager`, ale `Department má Manager`.

## 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 *Employee* i (přeneseně) z *Person*.

## Vícenásobná dědičnost

- Java vícenásobnou dědičnost ve smyslu dědění z více tříd současně **nepodporuje!**
- Důvodem je konflikt typu diamant:

```
class DoggyManager extends Employee, DogKeeper { }  
class Employee { String toString() { return "Employee"; } }  
class DogKeeper { String toString() { return "DogKeeper"; } }  
  
new DoggyManager().toString(); // we have a problem!
```



Vícenásobná dědičnost je možná jedině u rozhraní, kde metody nemají definovanou implementaci a konflikt implementací tak nenastává.

## Dědičnost a vlastnosti tříd

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

## Dědičnost vs. rozhraní

### Dědičnost

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

### Rozhraní

- méně závislostí, více případně i opakovaného kódu

- více používané v praxi

## Pravidla pro konstruktory podtříd

- Konstruktor musí volat vybraný konstruktor nadtřídy, nebo vlastní přetížený konstruktor (pomocí `this()`; s případnými parametry).
- Konstruktor nadtřídy se volá pomocí `super()`; s případnými parametry.
- Podobně jako u `this()`, volání `super()`; musí být první příkaz a může být pouze jedno.
- V konstrukturu nelze ani zkombinovat `super()` s `this()`.
- Pokud `super()` i `this()` chybí, volá se automaticky bezparametrický konstruktor nadtřídy (tj. vloží se `super()`). Ten ale musí existovat a být neprivátní.
- Obecně platí, že volaný konstruktor musí v nadtřídě existovat.

## Příklad s Account

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

## Nová třída CheckedAccount

Rozšíříme třídu `Account` tak, že bude mít minimální zůstatek.

```
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;
    }
    public CheckedAccount(Person owner) {
```

```
    this(owner, 0, 0);  
  }  
}
```

## Volání kódu nadtřídy

- Vylepšíme, aby zůstatek nebyl nižší než minimální zůstatek
- Realizujeme tedy změnu metody `debit` pomocí jejího *překrytí* (overriding)

```
public class CheckedAccount extends Account {  
  private int minimalBalance;  
  // constructors ...  
  @Override  
  public boolean debit(int amount) {  
    // check min. balance  
    if(getBalance() - amount >= minimalBalance) {  
      return super.debit(amount); // the debit is inherited  
    } else return false;  
  }  
}
```

## Volání kódu nadtřídy

```
public boolean debit(int amount) {  
  if(getBalance() - amount >= minimalBalance) {  
    return super.debit(amount); // the debit is inherited  
  } else return false;  
}
```

- 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!
- Takto lze volat jen bezprostředního předka.
- Něco jako `super.super.debit(amount)` je syntaktická chyba.

## Repl.it demo k dědičnosti - bankovní účty

- <https://replit.com/@tpitner/PB162-Java-Lecture-05-inheritance-Account>

## Pozor na překrývání atributů

- Nepoužívejte chráněné (`protected`) atributy. Je to "bad practice".

- Používejte zásadně privátní atributy s veřejnými nebo chráněnými gettery a settery.
- **Nikdy nepřekrývejte atributy!** Tj. podtřída nesmí nikdy obsahovat atribut se jménem, jako má některá z jejích nadtříd. Přesto, že je to syntakticky možné.

## Not what we wished to do

```
public class CheckedAccount extends Account {
    private int minimalBalance;
    private int balance; // CHYBA - PREKRYTI ATRIBUTU Z NADRIDY
    // konstruktors ...
    @Override
    public boolean debit(int amount) {
        // check min. balance
        if(getBalance() - amount >= minimalBalance) {
            return super.debit(amount); // the debit is inherited
        } else return false;
    }
}
```

## Namísto dědění lze použít skládání (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)
}
```

## Kompozice pořádně

- Kompozicí se zabývá navazující kurz *PV168 Seminář Javy*.
- Problémy s hierarchiemi dědičnosti pomocí kompozice řeší některé návrhové vzory, např.
  - **Bridge**: Řešení exploze podtříd rozdělením problémové domény na abstrakci a implementaci

- **Decorator**: Variantnost chování je přesunuta z podtříd do spolupráce několika malých objektů za běhu
- Více v magisterském kurzu *PA103 Object-oriented Methods for Design of Information Systems*

## Replit.com demo k dědičnosti - geometrické útvary

- <https://replit.com/@tpitner/PB162-Java-Lecture-05-inheritance-Shapes>