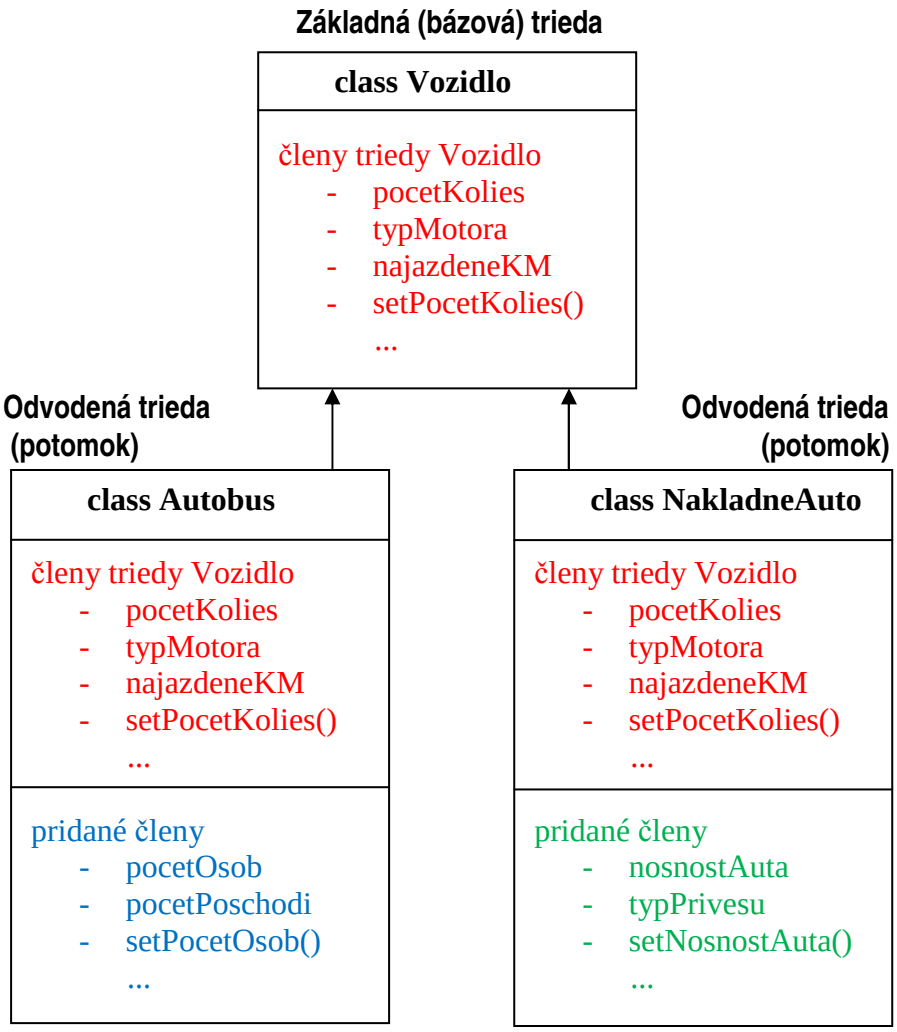
# 25 Dedičnosť

## 25.1 Dedičnosť (inheritance)

Jednou z najcharakteristickejších a najdôležitejších vlastností objektovo orientovaného programovania je **dedičnosť (inheritance)**. Jednoducho povedané dedičnosť v OOP znamená, že trieda môže dediť členy inej triedy. Triedu teda môžeme odvodiť z inej triedy.

Dôvodom na používanie dedičnosti je skutočnosť, že objekty majú niektoré vlastnosti rovnaké. Napríklad autobus a nákladné auto majú kolesá, majú motor, najazdia určitý počet kilometrov, spotrebujú isté množstvo paliva. Niektoré vlastnosti majú ale odlišné - oba spomínané dopravné prostriedky sú určené na prepravu – autobus na prepravu osôb, nákladné auto na prepravu tovaru. V tomto prípade by sme mohli vytvoriť triedu *Vozidlo*, ktorá by obsahovala spoločné stavy a schopnosti a dedením ďalšie dve triedy: triedu *Autobus* (zdedila by všetky členy triedy Vozidlo a doplnila svoje vlastné) a triedu *NakladneAuto* (zdedila by všetky členy triedy Vozidlo a doplnila svoje vlastné).




Triedu, z ktorej sa dedí, budeme nazývať **základnou**, **rodičovskou** alebo tiež **bázovou** triedou. Triedu, ktorá vznikne dedením, budeme nazývať **odvodenou** triedou alebo jednoducho **potomkom**.

Odvodená trieda môže byť tiež základnou, ak z nej bude dediť iná trieda. Takto sa dá vytvoriť celá vlastná **hierarchia tried**.

Aj štandardné triedy Javy vytvárajú hierarchiu, ktorej strom možno vidieť napríklad na stránke: http://docs.oracle.com/javase/7/docs/api/java/lang/package-tree.html

V Jave je povolená iba tzv. jednoduchá dedičnosť, kedy trieda vzniká dedením iba z jednej bázovej triedy. Viacnásobnú dedičnosť (známu a často využívanú v C++) možno v Jave realizovať pomocou **rozhraní** – rozhrania si vysvetlíme neskôr.

Všetky triedy v Jave – aj tie, ktoré si vytvárame my sami – sú potomkami triedy **Object**. Trieda *Object* je jedinou triedou v Jave, ktorá nemá žiadneho predchodcu.

## 25.2 Rezervované slovo `extends`

Využitím rezervovaného slova **extends** môžeme zo základnej triedy vytvoriť odvodenú triedu:

```
public class OdvodenaTrieda extends ZakladnaTrieda
{
    //Pridáme členy odvodenej triedy
}
```

Trieda *OdvodenaTrieda* vznikla dedením z triedy *ZakladnaTrieda*. *OdvodenaTrieda* zdedí všetky členy triedy (okrem konštruktorov!) *ZakladnaTrieda* a doplní svoje vlastné členy, prípadne zmení funkčnosť niektorých metód základnej triedy.

Použitie dedičnosti si ukážeme na nasledovnej aplikácii.

## 25.3 Aplikácia Dedicnost

Vytvoríme aplikáciu s názvom **Dedicnost**, na ktorej si ukážeme všetky podstatné rysy dedenia. Opäť sa budeme zaoberať vozidlami. Vytvoríme si základnú triedu Vozidlo – deklarujeme v nej jednu súkromnú členskú premennú s názvom *pocetKolies*. Konštruktory zatiaľ neimplementujeme, ale prístupové metódy áno. Základná trieda Vozidlo by mohla vyzerať nasledovne:

```
public class Vozidlo
{
 private int pocetKolies;

 public void setPocetKolies(int pocetKolies)

 {
    this.pocetKolies = pocetKolies;
 }
```

```
 public int getPocetKolies()
 {
    return this.pocetKolies;
 }
}
```

Teraz vytvoríme odvodenú triedu *Autobus*, ktorá bude dediť z triedy *Vozidlo*, doplníme do nej súkromnú premennú *pocetOsob* aj s prístupovými metódami:

```
public class Autobus extends Vozidlo
{
 private int pocetOsob;

 public void setPocetOsob(int pocetOsob)                    //
 {
    this.pocetOsob = pocetOsob;
 }

 public int getPocetOsob()
 {
    return this.pocetOsob;
 }
}
```

Ďalej vytvoríme odvodenú triedu *NakladneAuto*, ktorá bude dediť z triedy *Vozidlo*, doplníme do nej súkromnú premennú *nosnostAuta* aj s prístupovými metódami:

```
public class NakladneAuto extends Vozidlo
{
 private int nosnostAuta;

 public void setNosnostAuta(int nosnostAuta)
 //
 {
    this.nosnostAuta = nosnostAuta;
 }

 public int getNosnostAuta()
 {
    return this.nosnostAuta;
 }
}
```

Máme všetko potrebné k tomu, aby sme si mohli vytvoriť inštancie odvodených tried a vyskúšať dedičnosť. V metóde *main()* zapíšeme nasledovné príkazy:

```
Autobus autobus1 = new Autobus();
autobus1.setPocetKolies(10);    //volaná metóda z rodičovskej triedy
autobus1.setPocetOsob(40);      //volaná metóda z triedy Autobus
```

```
System.out.println(autobus1.getPocetKolies());
System.out.println(autobus1.getPocetOsob());

NakladneAuto nakladneAuto1 = new NakladneAuto();
nakladneAuto1.setPocetKolies(10);    //volaná metóda z rodičovskej triedy
nakladneAuto1.setPocetOsob(40);      //volaná metóda z triedy NakladneAuto
```

Obe triedy majú svoje dve vlastné premenné, pričom premennú *pocetKolies* zdedili z rodičovskej triedy Vozidlo.

K premennej *pocetKolies* neexistuje priamy prístup, lebo je to súkromná členská premenná, ale k dispozícii sú prístupové metódy, ktoré sme aj využili.

Teraz do tried doplníme nasledovné členy:

- do triedy *Vozidlo*: premenné *typMotora* a *najazadeneKM* a prístupové metódy
- do triedy *Autobus*: premenné *pocetOsob* a *pocetPoschodi* a pristupové metódy
- do triedy *NakladneAuto*: premenné *nosnostAuta* a *typPrivesu* a prístupové metódy

Do metódy *main()* doplníme príkazy na zápis hodnôt do príslušných premenných a ich výpis.

V ďalšej časti sa pozrieme na konštruktory.

## 25.4 Konštruktory a dedičnosť.

Konštruktory sa nededia! Toto je najdôležitejšia informácia týkajúca sa konštruktorov v súvislosti s dedením. Ak vytvoríme konštruktor v potomkovi, tak nastavíme hodnoty členských premenných potomka tak, ako to zadáme v konštruktore potomka. Rodičovské premenné však budú inicializované na hodnoty, ktoré boli zadané v konštruktore rodiča. Ukážeme si to na konkrétnom prípade. Budeme pokračovať v aplikácii *Dedicnost*.

Do triedy Vozidlo vložíme bezparametrický konštruktor, ktorý nastaví hodnoty premenných napríklad nasledovne:

```
...
Vozidlo()
{
 this.pocetKolies = 4;
 this.typMotora = “benzin”;
 this.najazdeneKM = 100;
}
...
```

V metóde main() teraz vytvoríme novú inštanciu triedy *NakladneAuto* s názvom *nakladneAuto2* :

```
NakladneAuto nakladneAuto2 = new NakladneAuto();
```

Zaujíma nás ako boli volané konštruktory. Najskôr sa zavolá konštruktor rodičovskej triedy Vozidlo a nastaví hodnoty premenných tak, ako je to uvedené v konštruktore. Potom sa zavolá implicitný konštruktor (keďže sme svoj vlastný nevytvorili) triedy *NakladneVozidlo* a nastaví hodnoty tak, ako to už pri implicitných koštruktoroch poznáme – celočíselné premenné na 0, reálne na 0.0 atď. To, že pri konštrukcii objektu boli zavolané oba konštruktory sa presvedčíme výpisom počtu kolies a nosnosti auta:

```
System.out.println(nakladneAuto2.getPocetKolies()); // 4
System.out.println(nakladneAuto2.getNosnostAuta()); // 0.0
```

Teraz si vytvoríme vlastný konštruktor aj v triede *NakladneAuto*:

```
...
NakladneAuto()
{
  this.nosnostAuta = 6000;
  this.typPrivesu = “prepravník”;
}
...
```

Program skompilujme a spustime. Opäť dajme vypísať počet kolies a nosnosť auta:

```
System.out.println(nakladneAuto2.getPocetKolies()); // 4
System.out.println(nakladneAuto2.getNosnostAuta()); // 6000
```

Všetko funguje tak ako má. Najskôr bol zavolaný konštruktor *Vozidlo()*, potom konštruktor NakladneAuto (nami vytvorený).

Zdá sa, že všetko je v poriadku. Áno, ale len dovtedy, kým nebudeme požadovať konštrukciu objektu triedy NakladneAuto s takými hodnotami členských premenných rodičovskej triedy Vozidlo, aké zadáme pri konštrukcii objektu. Inak povedané, chceme napr. skonštruovať objekt *nakladnyAutomobil3* s počtom kolies 10, s typom motora „diesel“ a s počtom najazdených kilometrov 500. Najskôr doplníme do triedy Vozidlo ďalší parametrický konštruktor:

```
  ...
Vozidlo(int pocetKolies,String typMotora,double najazdeneKM)
{
   this.pocetKolies = pocetKolies;
   this.typMotora = typMotora;
   this.najazdeneKM = najazdeneKM;
}
  ...
```

Teraz už potrebujeme len „drobnosť“, zavolať tento konštruktor z triedy NakladneAuto!

**Konštruktor z nadradenej triedy voláme vždy metódou super().** Táto metóda môže byť parametrická alebo bezparametrická, podľa toho ako je navrhnutý konštruktor v rodičovskej triede a v konštruktore potomka musí byť uvedená ako **prvá!!!**

Do triedy NakladneAuto doplníme konšturktor aj s metódou *super()*:

```
...
NakladneAuto(int pocetKolies,String typMotora,
             double najazdeneKM,int nosnostAuta,
             String typPrivesu)
{
 super(pocetKolies,typMotora,najazdeneKM); //je volaný konštruktor
                                           //triedy Vozidlo
 this.nosnostAuta = nosnostAuta;
 this.typPrivesu = typPrivesu;
}
...
```

Ktorýkoľvek konštruktor rodiča môže byť volaný z ktoréhokoľvek koštruktora potomka. V danom konštruktore môže byť volaný iba raz a aj to hneď ako prvý príkaz v konštruktore!

## 25.5 Predefinovanie metódy rodičovskej triedy - overriding

Dedením nezíska potomok len členov rodičovskej triedy ale aj možnosť zmeniť funkčnosť metód rodiča – hovoríme tomu **predefinovanie** alebo **overriding**. Aj predefinovanie metódy si ukážeme na konkrétnom príklade. V triede *Autobus* predefinujeme jednu z metód implementovanú v triede *Vozidlo*.

Do triedy *Vozidlo* doplníme ďalšiu členskú premennú s názvom *rokVyroby*. Tento krát ju ale deklarujeme ako chránenú – **protected**. Je to kvôli tomu, aby sme k nej mali z potomka priamy prístup.

Vytvoríme setter a getter:

```
public class Vozidlo
{
 ...
 protected int rokVyroby;
 ...
 public void setRokVyroby(int rokVyroby)
 {
   this.rokVyroby = rokVyroby;
 }
 public int getRokVyroby()
 {
   return this.rokVyroby;
 }
}
```

Vidíme, že metóda *setRokVyroby*() v triede *Vozidlo* umožňuje zadať akýkoľvek rok výroby. My však pre autobusy vyžadujeme, aby bol rok výroby v intervale <2000,2013>. Preto metódu *setRokVyroby()* teraz predefinujeme v triede *Autobus* nasledovne:

```
...
//Predefinovanie metódy setRokVyroby() triedy Vozidlo
public void setRokVyroby(int rokVyroby)
{
  if (rokVyroby>=2000 && rokVyroby<=2013)
  {
     this.rokVyroby = rokVyroby;
  }
  else
  {
     this.rokVyroby = 2000;
  }
}
```

Uvedeným kódom dosiahneme to, že v inštanciách triedy *Autobus*, pre ktoré budeme chcieť zapísať rok výroby, sa vždy bude volať metóda *setRokVyroby()* triedy *Autobus* a nie triedy *Vozidlo*. Presvedčíme sa o tom tak, že v metóde *main()* zapíšeme prílazy:

```
autobus1.setRokVyroby(1900);
System.out.println(autobus1.getRokVyroby()); //vypíše 2000
```

Potomok môže okrem predefinovania metódy predka metódu aj preťažiť. Stačí uviesť rovnaký názov metódy a iný počet alebo iné typy formálnych parametrov. Toto už je ale problematika nám dobre známa.

## 25.6 Zamedzenie možnosti predefinovania metódy.

Ak chceme zamedziť možnosti predefinovania metódy v potomkovi, tak v rodičovskej triede uvedieme v hlavičke metódy rezervované slovo **final**. Takto deklarované metódy nie je možné predefinovať ale len preťažiť.

## 25.7 Zamedzenie možnosti dediť

Pri vytváraní konkrétnej triedy chceme niekedy zamedziť možnosti dediť z tejto triedy. V takomto prípade tiež použijeme rezervované slovo **final**. Napríklad nechceme aby bolo možné dediť triedu Autobus, tak ju definujeme nasledovne:

```
final public class Autobus extends Vozidlo
{
  ...
}
```

## 25.8 Abstraktné metódy a abstraktné triedy

Abstraktné triedy a abstraktné metódy sú opakom finálnych tried a metód. Abstraktné metódy musia byť v potomkovi vždy predefinované, preto tieto metódy neobsahujú žiaden výkonný kód. Ak abstraktnú metódu v potomkovi nepredefinujeme, tak kompilátor vyhlási chybu.

Ak je v triede aspoň jedna abstraktná metóda, tak aj trieda musí byť deklarovaná ako abstraktná. Od abstraktných tried nie je možné vytvárať inštancie. Takéto triedy sú len akousi šablónou pre vytvorenie potomkov. Abstraktné metódy a triedy sa deklarujú použitím rezervovaného slova **abstract**.

## 25.9 Cvičenie

1. Vytvorte aplikáciu s názvom **Banka2**. Deklarujte triedu s názvom **Ucet**. V nej deklarujte dve chránené premenné *cisloUctu* a *stavNaUcte*. Vytvorte konštruktory a prístupové metódy. Ďalej vytvorte triedu **BeznyUcet** ako potomka triedy *Ucet*. Vytvorte konštruktor, ktorý umožní volať konštruktor triedy *Ucet*. Ďalej implementujte metódy na možnosť vkladu a výberu finančnej hotovosti. Tiež odsledujte počet výberov a počet vkladov. Vytvorte niekoľko inštancií triedy *BeznyUcet* a vyskúšajte vklady a výbery.
2. Doplňte do predchádzajúcej aplikácie ďalšiu triedu s názvom **SporiaciUcet** ako potomka triedy *Ucet*. Vytvorte konštruktor, ktorý umožní volať konštruktor triedy *Ucet*. Ďalej implementujte metódu na možnosť vkladu. Implementujte tiež metódu na výber z účtu, ale len v prípade povoleného počtu vkladov (napríklad výber bude možný iba po desiatich vkladoch) – ošetrite v metóde. Vytvorte niekoľko inštancií triedy *SporiaciUcet* a vyskúšajte vklady a výbery.

## 25.10 Otázky

1. Stručne vysvetlite dedičnosť v OOP.
2. Prečo sa do OOP zaviedla dedičnosť?
3. Čím sa líši trieda *Object* od všetkých ostatných tried Javy?
4. Ako voláme triedu, z ktorej sa dedí?
5. Čo je to jednoduchá dedičnosť?
6. Ktoré rezervované slovo používame pre odvodenie triedy od inej triedy?
7. Ako zabezpečíme aby bol z potomka zavolaný konštruktor rodičovskej triedy?
8. Kde presne uvádzame volanie konštruktora rodičovskej triedy?
9. Čo rozumieme pod predefinovaním metód? V čom sa líši od preťaženia?
10. Ako možno zamedziť predefinovaniu metódy?
11. Ako možno zamedziť dedeniu danej triedy?
12. Čo sú to abstraktné metódy a triedy?