Java pro začátečníky (12) - Dědičnost a polymorfismus

Tento díl bude zasvěcen dědičnosti a polymorfismu, což jsou jedny ze základních kamenů objektově orientovaného programování. Dědičnost nám umožní vytvářet podtypy jednotlivých tříd, zatímco pomocí polymorfismu dosáhneme volání vždy toho správného objektu.

Dědičnost

Jak již bylo řečeno v úvodu článku, dědičnost nám umožňuje vytvářet hierarchie tříd, ve kterých můžeme o libovolném uzlu říct, že je speciálním případem libovolného ze svých předků. V reálném světě bychom řekli, že člověk je typem savce, auto je dopravním prostředkem, případně že voda je druhem nápoje.

Z druhé strany si ale řekněme, co je špatně použitou dědičností. Nikdy nemůžeme dědit letadlo od racka z důvodu, že má také křídla. Také nemůžeme dědit židli od člověka z titulu, že má člověk má dvě nohy, židle čtyři, tudíž stačí v podtypu dvě nohy přidat a vše bude v pořádku.

Třída Object

S dědičností jsme se již setkali, aniž bychom o tom věděli. Říkali jsme si, že každá třída má určité metody, které jsou již definovány. Jmenovitě jsme si zmiňovali metody *equals* a *hashCode* pro porovnávání objektů a metodu *finalize*, která je zavolána jako poslední pomazání *garbage collectorem* před likvidací objektu. Tyto metody naše objekty implicitně zdědily z třídy *java.lang.Object* (*dokumentace*).

Protože *Object* implicitně dědí všechny třídy, které explicitně nerozšiřují jinou třídu, tak je tato třída kořenem hierarchie všech tříd v Javě.

Extends

Pro vytvoření podtypu stačí v hlavičce třídy ihned po jejím názvu uvést klíčové slovo *extends* a název rozšiřované třídy. Takto vytvořená třída zdědí všechny nesoukromé (včetně package friendly, je-li rozšiřující třída ve stejném balíčku) metody a třídní proměnné předka, které může znovu deklarovat a překrýt.

Naopak třída nezdědí soukromé a statické metody svého předchůdce, protože ty se vztahují pouze ke konkrétní třídě předka. Metody označené jako koncové (*final*) potomek sice zdědí, ale nemůže je překrýt.

Specifikátory přístupu

Každý objekt potomka můžeme přetypovat na předka. Z toho plyne, že při překrývání metod můžeme jejich viditelnost pouze uvolňovat. Představme si, že by to takto nefungovalo a zpřísnili bychom přístup v potomkovi z *public* na *private*. Pak by nám stačilo objekt potomka přetypovat na předka, který má veřejný přístup a zavolat tuto v potomkovi soukromou metodu.

Vícenásobná dědičnost

V Javě můžeme dědit vždy maximálně od jednoho objektu. Toto je rozdíl oproti jiným objektově orientovaným jazykům, které často toto omezení nemají. Je ale třeba říci, že se do situace, ve které potřebujeme vícenásobnou dědičnost dostaneme pouze velmi zřídka, jelikož Java má kromě klasické dědičnosti ještě tzv. rozhraní (*interface*), kterých třída může realizovat libovolné množství (ale o nich až v dalším dílu).

Super

Při jednotlivých voláních metod podtypů často nazazíme na to, že nechceme celou metodu překrýt, pouze k ní chceme přidat další funkcionalitu. V tento okamžik můžeme zavolat *super.jmenoMetody()*, čímž zavoláme funkcionalitu předka. Analogicky k řetězení konstruktorů *this()* můžeme volat konstruktor předka voláním *super()* – toto volání musí být v rámci konstruktoru potomka vždy na prvním místě.

Dalším aspektem volání kostruktoru v rámci hierarchie je, že není příliš vhodné volat kteroukoliv metodu, která může být překryta. Při volání konstruktoru se objekt vytváří postupně z vrchu hierarchie (tzn. od třídy *Object*). Pokud bychom proto volali metodu, která je překrytá v některém z potomků, a která zde využívá fieldy, které ještě nemohly být inicializovány, tak bychom se mohli dočkat havárie programu.

Příklad

Mějme třídu zaměstnance (*jméno*, *věk*), která má pouze jednu metodu *work()*, jež identifikuje daného zaměstnance a vypíše, že pracuje (místo tohoto suchého výpisu si můžeme představit složitou aplikační logiku).

Z této třídy pak dědí třída ředitele. Ředitel obsahuje navíc kolekci podřízených zaměstnanců. V metodě *work()* tyto zaměstnance jmenuje (můžeme si představit, že na nich volá složitou aplikační logiku a skládá výsledky jejich práce).

Třída Employee

Java

```java
package jpz12.example1;

/**
 * Trida zamestnance
 * @author Pavel Micka
 */
public class Employee {
    protected String name;
    protected int age;

    public Employee(String name, int age) {
        this.name = name;
        this.age = age;
    }


    public void work(){
        System.out.println("Jsem zamestnanec " + name + " ( " + age + " )"
                + " a pracuji na dulezitem ukolu");
    }

    /**
     * @return the age
     */
    public int getAge() {
        return age;
    }

    /**
     * @param age the age to set
     */
    public void setAge(int age) {
        this.age = age;
    }

    /**
     * @return the name
     */
    public String getName() {
        return name;
    }

    /**
     * @param name the name to set
     */
    public void setName(String name) {
        this.name = name;
```

```
48.     }
49. }
```

Na třídě zaměstnance nenalezneme nic překvapivého. Definuje jeden konstruktor s dvěma parametry – což mj. znamená, že třída nemá implicitní bezparametrický konstruktor, ten bychom museli v případě potřeby dodefinovat.

Třídá má dále dva fieldy, které reprezentují věk a jméno zaměstnance. K těmto fieldům přistupujeme pomocí getterů a setterů, abychom tak lépe zakryli vnitřní implementaci třídy.

Samotná metoda work funguje přesně dle specifikace (vypíše jméno pracovníka, jeho věk a to, že usilovně pracuje).

Třída Director

Java

```java
package jpz12.example1;

/**
 * Trida reditele (zamestnance, ktery ma jeste specificky ukol)
 * @author malejpavouk
 */
public class Director extends Employee {
    private Employee[] employees;

    public Director(String name, int age, Employee[] employees) {
        super(name, age); //volame konstruktor predka
        this.employees = employees;
    }

    @Override //rikame prekladaci, ze prekryvame metodu
    public void work() {
        super.work(); //volame metodu work predka

        System.out.println("Prave ridim tyto lidi: ");
        for (int i = 0; i < employees.length; i++) {
            if (i != 0) { //pred prvnim zamestnancem neni ve vypisu carka
                System.out.print(", ");
            }
            System.out.print(employees[i].getName());
        }
        System.out.println(""); //nakonec odradkujeme
    }

    /**
     * @return the employees
     */
    public Employee[] getEmployees() {
        return employees;
    }

    /**
     * @param employees the employees to set
     */
    public void setEmployees(Employee[] employees) {
        this.employees = employees;
    }
}
```

Třída ředitele má přístup k fieldům age a name, jelikož byly specifikované jako *protected* (tj. mají zakázený přístup z vnějšího světa, ale povolený přístup z potomků). Mohli bychom je tedy nastavit přímo v konstruktoru třídy *Director*. Tím bychom ale zbytečně duplikovali kód, v případě změny bychom pak museli měnit kód na dvou místech, čímž bychom si zdvojnásobili šanci, že uděláme nějakou chybu. Proto v konstruktoru třídy *Director* nejprve zavoláme konstruktor předka

a poté pouze nastavíme zbylé fieldy (pole zaměstnanců).

V metodě *work()* nejprve zavoláme *work()* třídy zaměstnance, comž dojde k onomu výpisu. Poté pokračujeme v práci a vypíšeme podřízené zaměstnance.

@Override

Také si všimněme anotace *@Override*, pomocí které říkáme překladači, že překrýváme metodu. Její přítomnost není povinná, vše by fungovalo i bez ní, ale v případě, že bychom udělali typografickou chybu (a metodu v potomkovi nedopatřením pojmenovali jinak), tak překladač zjistí, že jsme chtěli překrývat, ale nepřekrýváme a program nepřeloží. Tímto dojde k odhalení špatně chyby, která je za určitých okolností poměrně špatně dohledatelná.

Volání programu

```
01. /**
02.  * @param args the command line arguments
03.  */
04. public static void main(String[] args) {
05.     Employee em1 = new Employee("Pepa Smetak", 18);
06.     Employee em2 = new Employee("Franta Prodavac", 20);
07.     Employee em3 = new Employee("Karel Opravar", 40);
08. 
09.     em1.work();
10.     em2.work();
11.     em3.work();
12. 
13.     Employee[] array = {em1, em2, em3}; //vytvorime pole tri zamestnancu
14. 
15.     Director dir = new Director("Petr Podnikatel", 35, array);
16.     dir.work();
17. }
```

```
1. Jsem zamestnanec Pepa Smetak ( 18 ) a pracuji na dulezitem ukolu
2. Jsem zamestnanec Franta Prodavac ( 20 ) a pracuji na dulezitem ukolu
3. Jsem zamestnanec Karel Opravar ( 40 ) a pracuji na dulezitem ukolu
4. Jsem zamestnanec Petr Podnikatel ( 35 ) a pracuji na dulezitem ukolu
5. Prave ridim tyto lidi:
6. Pepa Smetak, Franta Prodavac, Karel Opravar
```

Polymorfismus

Jak jsme si již několikrát řekli, tak můžeme libovolnou metodu překrýt v potomkovi vlastní implementací. Pokud pak nad daným objektem tuto metodu zavoláme, dojde vždy k vykonání překrývajícího kódu. A to bez ohledu na to, jestli k metodě přistupujeme pomocí reference na objekt předka nebo na objekt potomka (jehož třída obsahuje ono překrytí).

Této vlastnosti je dosaženo pomocí pozdní vazby (*late binding*), kdy je typ objektu, na němž bude metoda volána, rozhodnut až za běhu programu, nikoliv během jeho kompilace.

Toto ovšem platí pouze pro překrývání metod, nikoliv pro jejich přetěžování. Pokud budeme mít na daném objektu dvě metody, které se budou lišit pouze parametrem (jedna pro předka, druhá pro potomka), tak bude volána vždy ta metoda, která má v parametru stejný typ, jako je aktuální reference na objekt. Volání přetížených metod je totiž rozhodováno v době překladu, kdy ještě nemusí být jasné, jesli bude reference ukazovat na objekt předka nebo potomka.

Příklad

```
01. /**
02.  * @param args the command line arguments
03.  */
04. public static void main(String[] args) {
05.     Employee em1 = new Employee("Pepa Smetak", 18);
06.     Employee em2 = new Employee("Franta Prodavac", 20);
```

```
    Employee em3 = new Employee("Karel Opravar", 40);
    Employee[] array = {em1, em2, em3}; //vytvorime pole tri zamestnancu

    Director dir = new Director("Petr Podnikatel", 35, array);

    //Pole, ktere ma 4 ukazatele na predky
    Employee[] array2 = {em1, em2, em3, dir};

    //for each cyklus pro pruchod pres vsechny prvky kolekce (pole)
    for (Employee employee : array2) {
        /* metoda je vzdy zavolana na referenci typu predka,
         * ale i tak dojde k volani prekryte metody na objektu potomka (Director)
         */
        employee.work();
        /**
         * Zatimco u volani metody na objektu dojde vzdy k volani "spravne" metody,
         * tj. te prekryte, tak pri volani pretizene metody se dle parametru
         * rozhodne vzdy pro volani metody s parametrem korespondujicim pri prekladu
         * tj. v tomto pripade vzdy pro metodu s parametrem ukazatele na predka (bez
         * ohledu na to, ze se jedna o objekt potomka)
         */
        testOverloading(employee);
    }
}

private static void testOverloading(Employee employee) {
    System.out.println("Method with employee parameter");
}

private static void testOverloading(Director director) {
    System.out.println("Method with employee parameter");
}
```

```
Jsem zamestnanec Pepa Smetak ( 18 ) a pracuji na dulezitem ukolu
Method with employee parameter
Jsem zamestnanec Franta Prodavac ( 20 ) a pracuji na dulezitem ukolu
Method with employee parameter
Jsem zamestnanec Karel Opravar ( 40 ) a pracuji na dulezitem ukolu
Method with employee parameter
Jsem zamestnanec Petr Podnikatel ( 35 ) a pracuji na dulezitem ukolu
Prave ridim tyto lidi:
Pepa Smetak, Franta Prodavac, Karel Opravar
Method with employee parameter
```

– Pavel Mička
Zakladatel a administrátor stránek Algoritmy.net