Objektovo orientované programovanie

(trieda Object, literál .class, trieda Objects)

8. prednáška (1.časť)

Vladislav Novák
FEI STU v Bratislave
4.11.2014

Obsah

Metódy triedy Object 1
Metóda clone() 1
Metóda getClass() 5
Literál `.class` 6
Java Reflection 6
Metóda equals() 7
Metóda hashCode() 7
Metóda toString() 10
Metóda finalize() 11
Trieda Objects 11

Metódy triedy Object

Všetky triedy sú priamymi, alebo nepriamymi potomkami triedy `Object` v balíku `java.lang`. Tento balík sa importuje automaticky.
Trieda `Object` definuje niekoľko metód. Niektoré z nich možno prekryť kódom, ktorý bude špecifický pre danú triedu.
Dnes sa zoznámime s metódami:

```
protected Object clone() throws CloneNotSupportedException
public boolean equals(Object obj)
public int hashCode()
public String toString()
public final Class<?> getClass()
protected void finalize() throws Throwable
```

Ďalšie metódy (súvisia so synchronizáciou činností vykonávaných vo viacerých vláknach):

```
public final void notify()
public final void notifyAll()
public final void wait() throws InterruptedException
public final void wait(long timeout) throws InterruptedException
public final void wait(long timeout,int nanos) throws InterruptedException
```

Metóda clone()

```
protected Object clone() throws CloneNotSupportedException
```

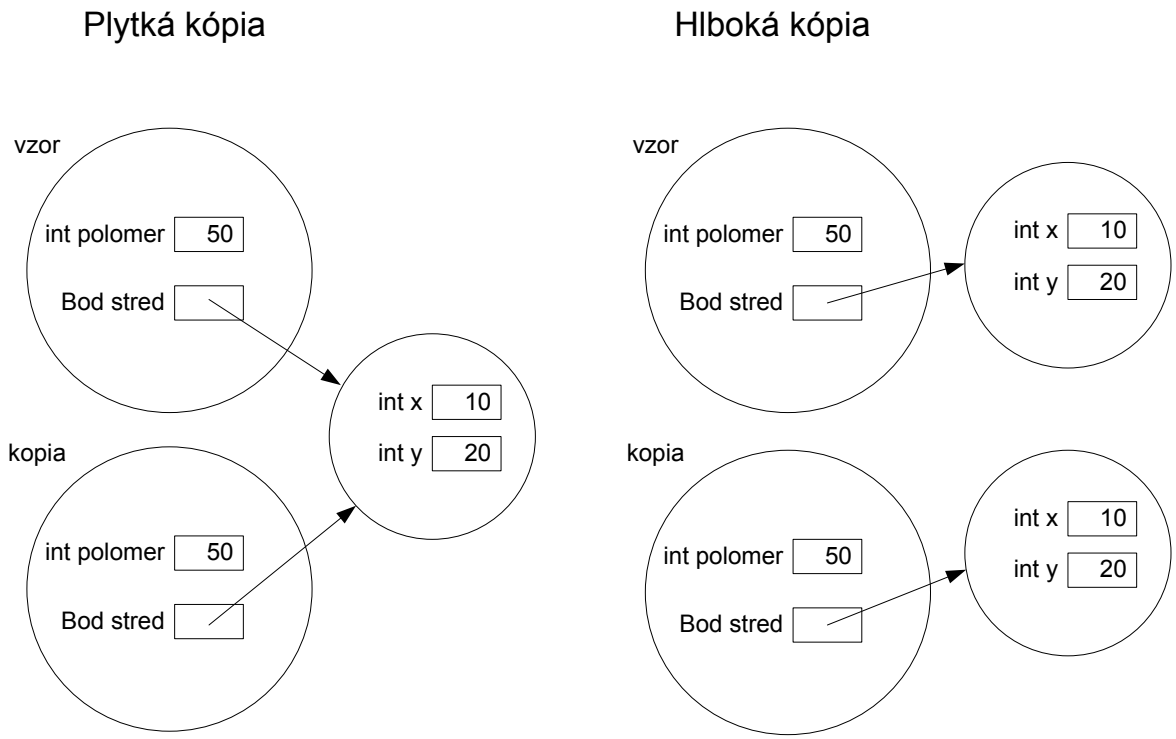
Ak trieda, alebo niektorá z jej nadtried implementuje rozhranie `Cloneable`, potom môžeme pomocou metódy `clone()` vytvoriť kópiu existujúceho objektu.

Implementácia tejto metódy v triede `Object` najprv kontroluje, či bola vyvolaná nad objektom ktorý implementuje rozhranie `Cloneable`.
Ak objekt toto rozhranie neimplementuje, tak vyhodí výnimku `CloneNotSupportedException`.
Ak objekt toto rozhranie implementuje, tak metóda vytvorí a vráti jeho kópiu (vráti nový objekt rovnakého typu s rovnakými hodnotami atribútov).
Trieda `Object` neimplementuje rozhranie `Cloneable`.
Rozhranie `Cloneable` neobsahuje žiadnu metódu (ani metódu `clone()`).

Rozoznávame 2 druhy vytvárania kópii objektov:
Plytká kópia objektu (shallow copy) – atribúty referenčného typu v originály aj v kópii odkazujú na tie isté objekty.
Hlboká kópia objektu (deep copy) – atribúty referenčného typu v origináli aj v kópii odkazujú na iné objekty, ktorých vnútorný stav je ale rovnaký.

V prípadoch keď potrebujeme vytvoriť plytkú kópiu objektu, je implementácia metódy v triede `Object` vyhovujúca. Ak chceme vytvoriť hlbokú kópiu, treba vytvoriť vlastnú implementáciu metódy `clone()`.

príklad (plytká a hlboká kópia objektov triedy kružnica):



príklad – plytká kópia:

```
public class Bod {
    private int x;
    private int y;

    public Bod(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

```
public class Kruznica implements Cloneable {
    private Bod stred;
    private int polomer;

    public Kruznica(Bod stred, int polomer) {
        this.stred = stred;
        this.polomer = polomer;
    }

    @Override
    public Kruznica clone() throws CloneNotSupportedException {
        return (Kruznica) super.clone();
    }
}
```

Metódu `clone()` nie je potrebne prekrývať, stačí implementovať rozhranie `Cloneable`. Bez prekrytia zostane metóda `clone() protected` a návratový typ bude `Object` (čo môže spôsobiť, že pri použití metódy bude potrebné explicitné pretypovanie).

príklad – hlboká kópia

```
public class Bod implements Cloneable{
    private int x;
    private int y;

    public Bod(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int dajX() {
        return x;
    }

    public int dajY() {
        return y;
    }

    public void nastav(int x, int y) { //je možné meniť stav objektu
        this.x = x;
        this.y = y;
    }

    @Override
    public Bod clone() throws CloneNotSupportedException {
        return (Bod) super.clone(); //atribúty sú základných typov
    }
}
```

```
public class Kruznica implements Cloneable {
    private Bod stred;
    private int polomer;

    public Kruznica(Bod stred, int polomer) {
        this.stred = new Bod(stred.dajX(), stred.dajY());
        this.polomer = polomer;
    }

    @Override
    public Kruznica clone() throws CloneNotSupportedException {
        Kruznica kopia = (Kruznica) super.clone();
        kopia.stred = (Bod) stred.clone(); //referenčný, nie je immutable
        return kopia;
    }
```

Podľa konvencie by mala byť kópia objektu získaná volaním `super.clone()`. Toto volanie však vráti iba plytkú kópiu. Ak chceme získať hlbokú kópiu a objekt obsahuje atribúty referenčného typu ktorých stav sa môže meniť (nie sú immutable), tak je potrebné vytvoriť kópie týchto atribútov (napr. `atribút.clone()`) a nastaviť hodnoty (referencie) týchto atribútov na vytvorené kópie.

Takýto spôsob implementácie ale neumožňuje definovať metódu `clone()` v prípade, že niektoré atribúty sú `final`. Vtedy je nutné použiť iný spôsob implementácie, napríklad:

```
//definícia triedy Bod môže byť rovnaká, ako v predchádzajúcom príklade

public class Kruznica implements Cloneable {
    private final Bod stred; //teraz je atribút stred final
    private int polomer;

    public Kruznica(Bod stred, int polomer) {
        this.stred = stred;
        this.polomer = polomer;
    }

    @Override
    public Kruznica clone() throws CloneNotSupportedException {
        return
         new Kruznica(new Bod(stred.dajX(), stred.dajY()), polomer);
    }
}
```

Výhodou vytvárania kópie objektu pomocou metódy `clone()` oproti volaniu konštruktora môže byť to, že nemusíme zadávať názov triedy. Čiže môžeme vytvoriť kópiu objektu bez toho aby sme presne poznali triedu, ktorej je inštanciou.

Pomocou metódy `clone()` možno vytvárať plytké kópie polí.

Metóda getClass()

```
public final Class<?> getClass()
```

Metódu `getClass()` nemožno prekryť. Táto metóda vráti objekt typu `Class`, ktorý umožňuje získať informácie o triede.

príklad:

```
public class Bod {
    private int suradnicaX;
    private int suradnicaY;

    public Bod() {
        this(0, 0);
    }

    public Bod(int suradnicaX, int suradnicaY) {
        this.suradnicaX = suradnicaX;
        this.suradnicaY = suradnicaY;
    }

    public String toString() {
        return "(" + suradnicaX + ", " + suradnicaY + ')';
    }
}
```

```
import java.lang.reflect.Field;

public static void main(String[] args) throws Exception {
    Bod b1 = new Bod(10, 20);

    System.out.println(b1.getClass().getSimpleName());
    System.out.println(b1.getClass().getSuperclass().getSimpleName());
    System.out.println(b1.getClass().getSuperclass().getName());
    System.out.println();

    for (Field atribut : b1.getClass().getDeclaredFields()) {
        System.out.println(atribut.getName());
    }
    System.out.println();

    Bod b2 =
     b1.getClass().getConstructor(int.class, int.class).newInstance(11,22);
    System.out.println(b2);

    Bod b3 = b1.getClass().newInstance();
    System.out.println(b3);
}
```

výstup:

```
Bod
Object
java.lang.Object

suradnicaX
suradnicaY

(11, 22)
(0, 0)
```

Literál `.class`

Podobne môžeme získať informácie o triede pomocou názvu triedy a literálu `.class`

príklad (definícia triedy `Bod` z predchádzajúceho príkladu):

```
public static void main(String[] args) throws Exception {
    System.out.println(Bod.class.getSimpleName());
    System.out.println(Bod.class.getSuperclass().getSimpleName());
    System.out.println(Bod.class.getSuperclass().getName());
    System.out.println();

    for (Field atribut : Bod.class.getDeclaredFields()) {
        System.out.println(atribut.getName());
    }
    System.out.println();

    Bod b2 =
        Bod.class.getConstructor(int.class, int.class).newInstance(11, 22);
    System.out.println(b2);

    Bod b3 = Bod.class.newInstance();
    System.out.println(b3);
}
```

výstup:

```
Bod
Object
java.lang.Object

suradnicaX
suradnicaY

(11, 22)
(0, 0)
```

Java Reflection

Java Reflection umožňuje získavať informácie o typoch za behu aplikácie. Tiež umožňuje vytvárať nové inštancie, volať metódy, nastavovať hodnoty atribútov. To je možné vykonávať aj bez znalosti typov v čase kompilácie kódu využívajúceho java reflection.
Príklady na zisťovanie informácii o triede sú uvedené v častiach venujúcich sa metóde `getClass()` a literálu `.class`.

Metóda equals()
`public boolean equals(Object obj)`

Metóda porovnáva dva objekty. Ak sú rovnaké, vráti `true`, inak vráti `false`. Implementácia metódy v triede `Object` porovnáva referencie na objekty (pomocou operátora identity ==). Ak chceme zistiť či sú dva objekty v rovnakom stave (či majú rovnaké hodnoty atribútov), potom treba túto metódu prekryť.

Ak je prekrytá metóda `equals()`, potom je potrebné prekryť aj metódu `hashCode()`.

Metóda hashCode()
`public int hashCode()`

Metóda `hashCode()` vracia hešovací kód objektu. Vlastnosti hešovacieho kódu: Ak sú dva objekty rovnaké, potom musia mať aj rovnaké hešovacie kódy. Ak sú dva objekty rôzne, potom majú hešovacie kódy s veľkou pravdepodobnosťou rôzne.
Prekrytie metódy `equals()`, zmení spôsob porovnávania objektov. Preto ak prekryjeme metódu `equals()`, treba prekryť aj metódu `hashCode()`.

príklad prekrytia metód `equals` a `hashCode`:

```
public class Bod {
    private int x;
    private int y;

    public Bod(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int dajX() {
        return x;
    }

    public int dajY() {
        return y;
    }

    @Override
    public boolean equals(Object obj) {
        if (obj == null) {
            return false;
        }
        if (getClass() != obj.getClass()) { //test či sú this a obj rovnakého typu
            return false;
        }
        final Bod other = (Bod) obj;
        if (this.x != other.x) {
            return false;
        }
        if (this.y != other.y) {
            return false;
        }
        return true;
    }
```

```
    @Override
    public int hashCode() {
        int hash = 7;
        hash = 53 * hash + this.x;
        hash = 53 * hash + this.y;
        return hash;
    }
}
```

```
public class Kruznica {
    private Bod stred;
    private int polomer;

    public Kruznica(Bod stred, int polomer) {
        this.stred = new Bod(stred.dajX(), stred.dajY());
        this.polomer = polomer;
    }

    @Override                    //ďalej bude uvedený jednoduchší spôsob implementácie
    public boolean equals(Object obj) {
        if (obj == null) {
            return false;
        }
        if (getClass() != obj.getClass()) {
            return false;
        }
        final Kruznica other = (Kruznica) obj;
        if (this.stred != other.stred && (this.stred == null ||
                                !this.stred.equals(other.stred))) {
            return false;
        }
        if (this.polomer != other.polomer) {
            return false;
        }
        return true;
    }

    @Override
    public int hashCode() { //ďalej bude uvedený jednoduchší spôsob implementácie
        int hash = 7;
        hash = 17 * hash +
                   (this.stred != null ? this.stred.hashCode() : 0);
        hash = 17 * hash + this.polomer;
        return hash;
    }
}
```

```
public static void main(String[] args) {
    Bod b1 = new Bod(10, 20);
    Bod b2 = new Bod(10, 20);
    Bod b3 = new Bod(7, 8);

    Kruznica k1 = new Kruznica(b1, 5);
    Kruznica k2 = new Kruznica(b2, 5);
    Kruznica k3 = new Kruznica(b3, 5);

    System.out.println(b1.equals(b2)); //true
    System.out.println(b1.equals(b3)); //false

    System.out.println(b1.hashCode()); //20213
    System.out.println(b2.hashCode()); //20213
    System.out.println(b3.hashCode()); //20042

    System.out.println(k1.equals(k2)); //true
    System.out.println(k1.equals(k3)); //false
}
```

Využitím triedy `Objects` je implementácia metód `equals(Object)` a `hashCode()` v triede `Kruznica` jednoduchšia. Trieda `Objects` je stručne popísaná nižšie.

```
public class Kruznica {
    private Bod stred;
    private int polomer;

    public Kruznica(Bod stred, int polomer) {
        this.stred = new Bod(stred.dajX(), stred.dajY());
        this.polomer = polomer;
    }

    @Override
    public boolean equals(Object obj) {
        if (obj == null) {
            return false;
        }
        if (getClass() != obj.getClass()) {
            return false;
        }
        final Kruznica other = (Kruznica) obj;
        if (!Objects.equals(this.stred, other.stred)) {
            return false;
        }
        if (this.polomer != other.polomer) {
            return false;
        }
        return true;
    }

    @Override
    public int hashCode() {
        int hash = 7;
        hash = 17 * hash + Objects.hashCode(this.stred);
        hash = 17 * hash + this.polomer;
        return hash;
    }
}
```

Metóda toString()

```
public String toString()
```

Metóda `toString()` vracia textovú reprezentáciu objektu. Táto metóda je napr. užitočná pri ladení programu. Využívajú ju niektoré metódy. Napríklad volanie `System.out.println(Object obj)` zavolá metódu `String.valueOf(obj)`, ktorá v prípade že `obj` nie je `null` zavolá metódu `obj.toString()` a reťazec vrátený metódou `toString()` sa vytlačí.

príklad:

```
public class Bod {
    private int x;
    private int y;

    public Bod(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public String toString() {
        return "(" + x + ", " + y + ")";
    }
}
```

```
public static void main(String[] args) {
    Bod b1 = new Bod(10, 20);
    System.out.println(b1.toString()); //"(10, 20)"
    System.out.println(b1);            //"(10, 20)"
}
```

Metóda finalize()

```
protected void finalize() throws Throwable
```

Túto metódu môže volať automatická správa pamäti (garbage collector) ak objekt už nie je používaný (žiadna premenná neobsahuje referenciu na objekt). Táto metóda sa môže použiť aby si objekt „po sebe poupratoval“ napr. uvoľnením používaných prostriedkov.
Nie je ale zaručené, že túto metódu systém zavolá.
Implementácia metódy `finalize()` v triede `Object` nemá žiadny efekt.
Prekrytá metóda `finalize()` by mala na konci volať metódu `finalize()` nadtriedy.

príklad:

```
public class Bod {
    private int x;
    private int y;

    public Bod(int x, int y) {
        this.x = x;
        this.y = y;
    }

    protected void finalize() throws Throwable{
        System.out.println("metoda finalize");
        super.finalize();
    }
}
```

```
public static void main(String[] args) {
    Bod bod = new Bod(10, 20);
    bod = null;
    System.gc(); //volanie odporučí JVM vymazať nepoužívané objekty z pamäte
}
```

Volanie `System.gc()` je jeden zo spôsobov ako odporučiť JVM uvoľnenie nepoužívaných objektov z pamäte. Volanie nezaručuje uvoľnenie objektov z pamäte.

Trieda Objects

Trieda `Objects` obsahuje niekoľko užitočných statických metód. Výhodou metód môže byť ošetrovanie použitia s nulovými referenciami. Napr. metóda

```
public static boolean Objects.equals(Object a, Object b)
```

vráti `true` ak sú argumenty rovnaké a `false` ak sú argumenty rôzne (podobne ako metóda `equals` definovaná v triede `Object`). Ak sú obidva argumenty `null`, potom metóda vráti `true`. Ak jeden argument je `null` a druhý nie je, potom vráti `false`. Rovnakosť je testovaná volaním metódy `Object.equals(Object)` nad prvým argumentom. Pri použití metódy `Objects.equals(Object a, Object b)` nemusíme zapisovať ošetrenie prípadu keď by sme volali metódu nad nulovou referenciou

implementácia `Objects.equals(Object a, Object b)`

```
public static boolean equals(Object a, Object b) {
        return (a == b) || (a != null && a.equals(b));
}
```