Objektovo orientované programovanie

(dedičnosť, pretypovanie, instanceof, prekrývanie metód, super, finálne metódy, finálne triedy, volanie prekrytej metódy v konštruktore, abstraktné metódy, abstraktné triedy, Liskovej princíp substitúcie, pravidlo „je“ („is a“))

6. prednáška

Vladislav Novák
FEI STU v Bratislave
21.10.2014

Obsah

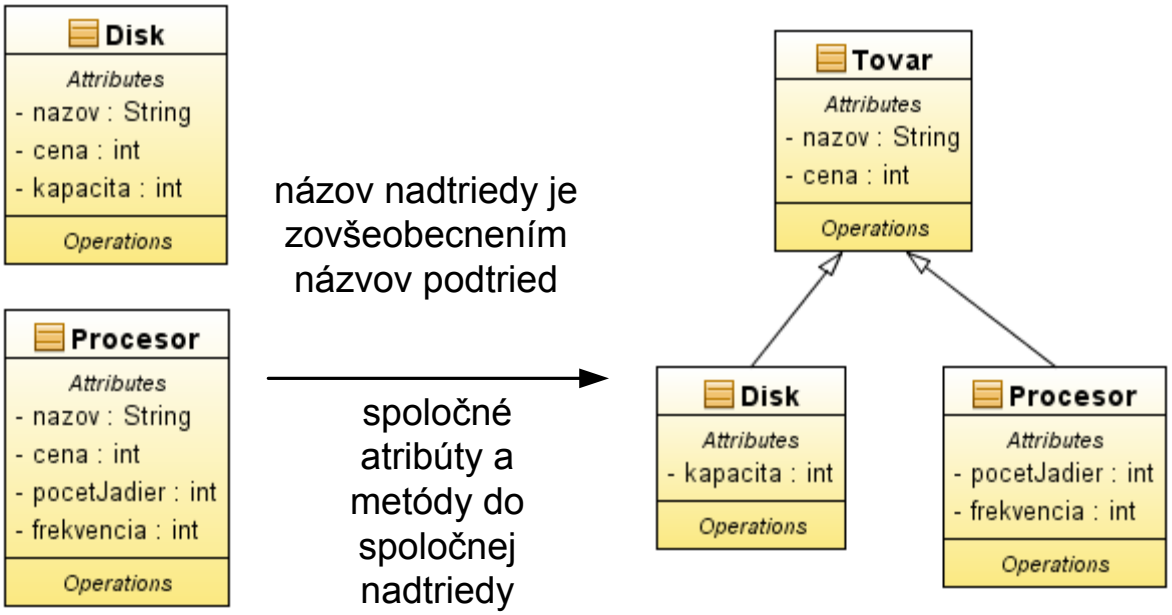
Dedičnosť (inheritance)

Príklad
Nech triedy `Disk` a `Procesor` reprezentujú druhy predávaných tovarov v obchode, pre ktorý vytvárame informačný systém. Obidve triedy možno zovšeobecniť ako tovary. Dôsledkom často bývajú rovnaké časti kódu v obidvoch triedach, čo nastáva aj v tomto príklade. Mechanizmus dedičnosti umožňuje vytvoriť triedu `Tovar`, ktorá bude obsahovať spoločné časti kódu obidvoch tried a následne vytvoriť triedy `Disk` a `Procesor` ako rozšírenia triedy `Tovar`. Hovoríme, že triedy `Disk` a `Procesor` sú odvodené od triedy `Tovar`, alebo dedia od triedy `Tovar`.

Trieda `Tovar` je zovšeobecnením tried `Disk` a `Procesor`, triedy `Disk` a `Procesor` sú špecializáciami triedy `Tovar`.

Trieda, ktorá je odvodená od inej triedy sa nazýva *podtrieda*, *odvodená trieda*, *rozšírená trieda*, *potomok*, alebo *podriadená trieda*. Trieda od ktorej sa podtrieda odvodzuje, sa nazýva *nadtrieda*, *základná trieda*, *rodič*, alebo *nadriadená trieda*.

Pri tvorbe odvodenej triedy je uvedené meno nadtriedy za kľúčovým slovom `extends`.



Nadtrieda Tovar:

```
public class Tovar { //trieda by mala byt abstraktná, ale o tom neskôr
    private String nazov;
    private int cena;

    public Tovar(String nazov, int cena) {
        this.nazov = nazov;
        this.cena = cena;
    }

    public int dajCenu() {
        return cena;
    }

    public void nastavCenu(int cena) {
        this.cena = cena;
    }

    public String dajNazov() {
        return nazov;
    }

    public void nastavNazov(String nazov) {
        this.nazov = nazov;
    }
}
```

podtrieda Disk:

```
public class Disk extends Tovar {
    private int kapacita; //v MB

    public Disk(String nazov, int cena, int kapacita) {
        super(nazov, cena); //konštruktoru nadtriedy odovzdáme parametre
        this.kapacita = kapacita;
    }

    public int dajKapacitu() {
        return kapacita;
    }

    public void nastavKapacitu(int kapacita) {
        this.kapacita = kapacita;
    }
}
```

Kľúčové slovo `super` sa v tomto príklade používa na volanie konštruktora nadtriedy (podobne ako `this` na volanie konštruktora tej istej triedy).

podtrieda Procesor:

```
public class Procesor extends Tovar {
    private int frekvencia;
    private int pocetJadier;

    public Procesor(String nazov, int cena,
                                  int frekvencia, int pocetJadier) {
        super(nazov, cena); //konštruktoru nadtriedy odovzdáme parametre
        this.frekvencia = frekvencia;
        this.pocetJadier = pocetJadier;
    }

    public int dajFrekvenciu() {
        return frekvencia;
    }

    public void nastavFrekvenciu(int frekvencia) {
        this.frekvencia = frekvencia;
    }

    public int dajPocetJadier() {
        return pocetJadier;
    }

    public void nastavPocetJadier(int pocetJadier) {
        this.pocetJadier = pocetJadier;
    }
}
```

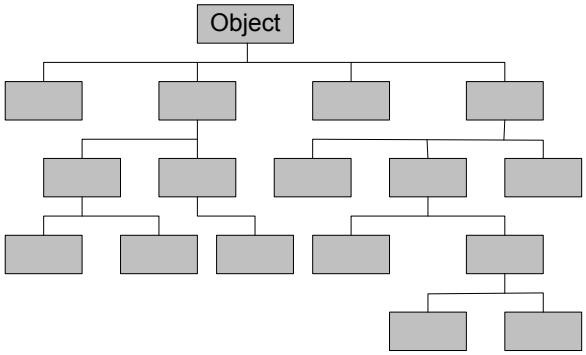
Podtrieda dedí (v podtriede možno používať) všetky členy (premenné, metódy, vnorené typy) s prístupovými právami `public` a `protected` bez ohľadu na balík v ktorom sa nachádzajú. Ak je podtrieda súčasťou rovnakého balíka ako nadtrieda, môže používať aj členy s prístupovými právami *package-private* (označenie bez modifikátora).

Podtrieda nededí členy nadtriedy, ktoré sú označené ako `private`. Tiež v nej nemožno priamo pristupovať ku členom nadtriedy bez modifikátora (package-private), ak sú nadtrieda a podtrieda v rôznych balíkoch.
Ak má napríklad nadtrieda zdedené metódy (označené ako `public`, alebo `protected`, alebo *package-private* v tom istom balíku), ktoré pracujú s jej členmi označenými ako `private`, alebo *package-private* v inom balíku, potom možno pomocou týchto metód nepriamo pristupovať k nezdedeným členom.
Podobne môže podtrieda pristupovať k nezdedeným členom aj cez zdedené vnorené triedy.

Prehľadnejšie to vyjadruje tabuľka v časti Prístupové práva na strane 9.

Konštruktory nepatria medzi členov, takže ich podtrieda nededí. Je však možné z podtriedy vyvolať konštruktor nadtriedy.

Každá trieda v Jave (okrem triedy `Object`) dedí od niektorej inej triedy. Ak pri vytváraní novej triedy neuvedieme jej nadtriedu, tak je automaticky nová trieda odvodená od triedy `Object`. Táto trieda je umiestnená najvyššie v hierarchii dedičnosti tried v jave. Každá trieda (okrem `Object`) priamo dedí od práve jednej triedy. Java neumožňuje viacnásobnú dedičnosť.



Trieda `Object` definuje a implementuje spoločné chovanie všetkých tried. Je definovaná v balíku `java.lang`.

Pretypovanie objektov (*casting*, *type casting*)

Pri prístupe k členom objektu zapisujeme
- názov premennej, ktorá obsahuje referenciu na objekt,
- bodku
- a názov člena objektu.

Kompilátor podľa typu premennej dokáže skontrolovať, či daný objekt obsahuje zadaný člen, alebo nie.
Mechanizmus dedenia zabezpečuje, že podtrieda obsahuje okrem členov v jej definícii, aj zdedené členy nadtriedy.

Implicitné pretypovanie
Majme triedu `T`. Premenná, ktorá je referenčnou premennou typu `T`, môže obsahovať nie len referencie na inštancie triedy `T`, ale aj na inštancie všetkých (priamych aj nepriamych) podtried triedy `T`.

Vráťme sa k príkladu Tovar, Disk, Procesor.
Nech kód programu obsahuje

```
Disk disk = new Disk("nazov",45,500);
Tovar tovarDisk = disk; //implicitné pretypovanie
```

Uvedený zápis neobsahuje chybu, pretože premenná `tovarDisk`, môže obsahovať referenciu nie len na inštanciu triedy `Tovar`, ale aj na inštancie jej podtried.
Zapis `Tovar tovarDisk = disk;` obsahuje pretypovanie, ktoré sa označuje ako *implicitné pretypovanie*.

Trieda `Tovar` aj všetky jej podtriedy obsahujú metódu `dajNazov()`, preto je nasledujúce volanie v poriadku

```
tovarDisk.getNazov();
```

Trieda `Tovar` ale neobsahuje metódu `dajKapacitu()`, preto kompilátor vyhodnotí nasledujúce volanie ako chybné, aj napriek tomu že premenná bude za behu referenciou na objekt ktorý metódu `dajKapacita()` obsahuje

```
tovarDisk.dajKapacitu(); //CHYBA
```

Kompilátor vo všeobecnosti nemôže vždy overiť, aký typ objektu bude za behu referencovaný danou premennou.

Explicitné pretypovanie

Majme triedu `T`. Referenčná premenná typu `T` nemôže vo všeobecnosti obsahovať referencie na inštancie nadtried typu `T`.
Zápis

```
Tovar tovar = new Tovar(„tovar", 10);
Procesor procesor = tovar; // CHYBA
```

je preto chybný. Kompilátor vo všeobecnosti nedokáže počas kompilácie kontrolovať volania metód nad objektmi referencovanými premennou `procesor`, pretože táto premenná môže za behu referencovať rôzne typy objektov.

Zápis

```
Tovar tovarProcesor = new Procesor("meno",100,2800,2);
Procesor proc2 = (Procesor) tovarProcesor;//explicitne pretypovanie
```

je v poriadku. Jeho druhý riadok obsahuje *explicitné pretypovanie*. Pri explicitnom pretypovaní kompilátor vloží do programu kontrolu, ktorá za behu programu skontroluje korektnosť pretypovania (či premenná `tovarProcesor` obsahuje inštanciu triedy Procesor)[1]. V prípade chyby (nemožnosti pretypovania) kontrola vyhodí výnimku za behu programu (výnimkami sa budeme zaoberať neskôr).

Zápis

```
proc2.dajNazov(); //zdedená metóda
proc2.dajFrekvenciu(); //metóda definovaná v triede Procesor
```

je OK.

Ak bude v kóde programu zápis

```
Procesor proc = (Procesor) tovar; //CHYBA ZA BEHU
```

nevznikne chyba pri kompilácii, ale vznikne chyba za behu programu.

[1] alebo inštanciu podtriedy triedy Procesor

Operátor `instanceof`

Testovanie typu objektu umožňuje operátor `instanceof`.
V nadväznosti na predchádzajúci príklad:

```
public static void main(String[] args) {
    Disk disk = new Disk("nazov",45,500);
    Tovar tovarDisk = disk; //implicitné pretypovanie
    Tovar tovar = new Tovar("tovar", 10);
    Tovar tovarProcesor = new Procesor("meno",100,2800,2);
    Procesor proc2 = (Procesor) tovarProcesor;//explicitne pret.

    System.out.println(disk instanceof Tovar);        //true
    System.out.println(disk instanceof Disk);         //true
  //System.out.println(disk instanceof Procesor); //CHYBA PRI KOMPILACII

    System.out.println(tovar instanceof Tovar);       //true
    System.out.println(tovar instanceof Disk);        //false
    System.out.println(tovar instanceof Procesor);    //false

    System.out.println(tovarDisk instanceof Tovar);   //true
    System.out.println(tovarDisk instanceof Disk);    //true
    System.out.println(tovarDisk instanceof Procesor);//false

    System.out.println(tovarProcesor instanceof Tovar); //true
    System.out.println(tovarProcesor instanceof Disk);  //false
    System.out.println(tovarProcesor instanceof Procesor);//true
}
```

Napr. objekt typu `Disk` je typu `Disk`, aj typu `Tovar`, aj typu `Object`, bez ohľadu na typ premennej.
Premenná `disk` nemôže obsahovať referenciu na inštanciu typu `Procesor`, preto kompilátor neumožní toto testovanie.

Operátor `instanceof` možno použiť aj na testovanie, či objekt implementuje určité rozhranie.

```
public interface RozhranieA { ..... }
```

```
public interface RozhranieB extends RozhranieA { .....}
```

```
public class ImplementaciaA implements RozhranieA{ ..... }
```

```
public class ImplementaciaB implements RozhranieB{ ..... }
```

```
public class Main {
    public static void main(String[] args) {
      ImplementaciaA a = new ImplementaciaA();
      ImplementaciaB b = new ImplementaciaB();
      Object o = new Object();

      System.out.println(a instanceof RozhranieA); //true
      System.out.println(a instanceof RozhranieB); //false
      System.out.println(b instanceof RozhranieA); //true
      System.out.println(b instanceof RozhranieB); //true
      System.out.println(o instanceof RozhranieB); //false
    }
}
```

Prekrývanie (override) metód inštancie

Ak má inštančná metóda v podtriede rovnakú signatúru a návratový typ ako inštančná metóda v nadtriede, tak inštančná metóda v podtriede *prekrýva* (*override*) inštančnú metódu v nadtriede.
Prekrývajúca metóda v podtriede môže tiež vracať podtyp typu, ktorý vracia metóda v nadtriede. Ide o *kovariantný návratový typ*.
Prekryté metódy sa označujú anotáciou `@Override`.

príklad:

```
public class TriedaA { /* ..... */}
```

```
public class TriedaB extends TriedaA { /* ..... */}
```

```
public class Nadtrieda {
    public int neprekryta(int a, int b) {
        System.out.println("Nadtrieda neprekryta(int,int)");
        return 2*(a+b);
    }

    public int prekryta1(int a, int b) {
        System.out.println("Nadtrieda prekryta1(int,int)");
        return a+b;
    }

    public TriedaA prekryta2(double a) {
        System.out.println("Nadtrieda prekryta2(double)");
        return new TriedaA();
    }

    public TriedaA prekryta3() {
        System.out.println("Nadtrieda prekryta3()");
        return new TriedaA();
    }
}
```

```
public class Podtrieda extends Nadtrieda{
    @Override
    public int prekryta1(int a, int b) {
        System.out.println("Podtrieda prekryta1(int,int)");
        return a*b;
    }

    @Override
    public TriedaA prekryta2(double a) {
        System.out.println("Podtrieda prekryta2(double)");
        return new TriedaB(); //implicitne pretypovanie
    }

    @Override
    public TriedaB prekryta3() { //kovariantny navratovy typ
        System.out.println("Podtrieda prekryta3()");
        return new TriedaB();
    }
}
```

```
public static void main(String[] args) {
    Nadtrieda nad = new Nadtrieda();
    Podtrieda pod = new Podtrieda();

    nad.neprekryta(1, 2); //Nadtrieda neprekryta(int,int)
    pod.neprekryta(1, 2); //Nadtrieda neprekryta(int,int)

    nad.prekryta1(1, 2); //Nadtrieda prekryta1(int,int)
    pod.prekryta1(1, 2); //Podtrieda prekryta1(int,int)

    nad.prekryta2(1); //Nadtrieda prekryta2()
    pod.prekryta2(1); //Podtrieda prekryta2()

    nad.prekryta3(); //Nadtrieda prekryta3()
    pod.prekryta3(); //Podtrieda prekryta3()

    //volanie virtuálnej metódy (popis nižšie)
    Nadtrieda akozeNad = pod;
    akozeNad.prekryta3(); //Podtrieda prekryta3()
}
```

Volanie virtuálnej metódy

Na konci predchádzajúceho príkladu je volaná metóda `prekryta3()`, kde pred bodkou je premenná typu `Nadtrieda`. Za behu však bude táto premenná obsahovať referenciu na objekt typu `Podtrieda`. Za behu sa zistí skutočný typ objektu nad ktorým sa metóda volá a podľa toho sa zavolá metóda z príslušnej triedy. Toto sa nazýva *volanie virtuálnej metódy* (*virtual method invocation*).
V C++ by sme museli metódu označiť ako virtuálnu, aby sa za behu skontroloval skutočný typ objektu a zavolala príslušná prekrývajúca metóda. V jave sú všetky metódy virtuálne.

Ukrývanie (hide) statických metód triedy

Ak je v podtriede definovaná statická metóda s rovnakou signatúrou ako v nadtriede, tak metóda v podtriede *ukrýva* príslušnú metódu z nadtriedy.

Ukrývanie členských premenných

Členská premenná v podtriede, ktorá má rovnaký názov ako členská premenná v nadtriede, ukryje príslušnú premennú z nadtriedy (aj vtedy ak sa líšia typom). Sú to dve rôzne členské premenné. Potom možno k členskej premennej z nadtriedy pristupovať pomocou kľúčového slova `super` (bude vysvetlené ďalej).
Ukrývanie členských premenných vo všeobecnosti komplikuje čitateľnosť kódu.

Modifikátory prístupu

Prístupové práva ku členom

Tabuľka prístupových práv (z prednášky o triedach a objektoch):

modifikátor	v rámci triedy	v inej triede toho istého balíka	v podtriede, ktorá je v inom balíku	v inej triede, iného balíka
`public`	ANO	ANO	ANO	ANO
`protected`	ANO	ANO	ANO	NIE
bez modifikátoru	ANO	ANO	NIE	NIE
`private`	ANO	NIE	NIE	NIE

`public` – člen môžeme používať v ktorejkoľvek časti programu
`protected` – člen môžeme používať
v triede kde je definovaný,
vo všetkých podtriedach
a triedach ktoré sú v tom istom balíku
bez modifikátoru – člen môžeme používať v celom balíku v ktorom je definovaný
`private` – člen môžeme používať iba v triede kde je definovaný

Zmena modifikátora prístupu

Pri prekrývaní metód môžeme zmeniť modifikátor prístupu k metóde. Prístupové práva môžeme rozšíriť, ale nemôžeme ich zmenšiť.
Kľúčové slovo `extends` naznačuje, že podtrieda je rozšírením nadtriedy. To znamená, že ak na niektorom mieste programu je priamo prístupná metóda inštancie nadtriedy, tak tam musí byť priamo prístupná aj prekrytá metóda inštancie podtriedy. Ak na niektorom mieste programu nie je priamo prístupná metóda inštancie nadtriedy, tak v podtriede pri prekrytí môžeme zmeniť jej prístupové práva tak, že na tomto mieste bude priamo prístupná prekrytá metóda inštancie podtriedy.

modifikátor v nadriede	**možný modifikátor v podtriede**
`private`	*tieto členy sa nededia*
package-private	package-private, `protected`, `public`
`protected`	`protected`, `public`
`public`	`public`

Použitie kľúčového slova `super`

Kľúčové slovo `super` sa používa na prístup k členom nadtriedy a pre volanie konštruktora nadtriedy. Jeho použitie je analogické s použitím kľúčového slova `this`.

Použitie `super` na prístup k členom nadtriedy

V členských metódach možno pristupovať k členom nadtriedy použitím kľúčového slova `super`. Využíva sa pri ukrytých členských premenných a prekrytých metódach.
Kľúčové slovo `super` je akoby referenciou na časť objektu definovanú nadtriedou, nad ktorým sa práve vykonáva členská metóda.

príklad:
Nadtrieda.java:

```
public class Nadtrieda {
    protected int atribut;

    public void metoda() {
        System.out.println("metoda v nadtriede");
    }
}
```

Podtrieda.java:

```
public class Podtrieda extends Nadtrieda {
    protected int atribut; //ukrýva členskú premennú z nadtriedy

    @Override
    public void metoda() {
        System.out.println("startujem v podtriede");
        super.metoda();
        this.atribut  = 10; //this možno vynechať
        super.atribut = 20;
        System.out.println("this.atribut="+this.atribut
                           +", super.atribut="+super.atribut);
    }
}
```

použitie:

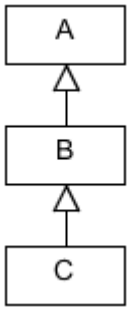
```
public static void main(String[] args) {
        Podtrieda pod = new Podtrieda();
        pod.metoda();
}
```

výstup:

```
startujem v podtriede
metoda v nadtriede
this.atribut=10, super.atribut=20
```

Volanie konštruktora nadtriedy

Pri vytváraní objektu, každý konštruktor zavolá najprv konštruktor nadtriedy a potom sa vykoná zvyšok daného konštruktora.



Príklad:
Nech sú definované triedy A, B, C také, že:
C dedí od B,
B dedí od A,
A nemá uvedenú nadtriedu, preto je jej nadtriedou trieda `Object`
Ak vytvoríme inštanciu triedy C, tak:
konštruktor triedy C na svojom začiatku vyvolá konštruktor triedy B,
konštruktor triedy B na svojom začiatku vyvolá konštruktor triedy A,
konštruktor triedy A na svojom začiatku vyvolá konštruktor triedy `Object`,
po dokončení konštruktora triedy `Object` sa vykoná zvyšok konštruktora triedy A
po dokončení konštruktora triedy A sa vykoná zvyšok konštruktora triedy B
po dokončení konštruktora triedy B sa vykoná zvyšok konštruktora triedy C
Toto sa nazýva *reťazenie konštruktorov*.

Konštruktor nadtriedy možno *vyvolať explicitne* pomocou kľúčového slova `super`. Ak konštruktor explicitne nevyvoláme, potom kompilátor doplní volanie konštruktora nadtriedy bez parametrov (*implicitné volanie konštruktora*).

Príklad:
Nadtrieda.java:

```
public class Nadrieda {
    private int atribut1;

    public Nadrieda(int atribut1) {
        this.atribut1 = atribut1;
    }

    public Nadrieda() {
        this(0);
    }
}
```

Podtrieda.java:

```
public class Podtrieda extends Nadrieda {
    private int atribut2;

    public Podtrieda(int atribut1, int atribut2) {
        super(atribut1);
        this.atribut2 = atribut2;
    }

    public Podtrieda(int atribut2) {
        super(); //mozeme vynechat (kompilator dokaze doplnit)
        this.atribut2 = atribut2;
    }

    public Podtrieda() { //implicitné volanie konštruktora nadtriedy
        this(0);
    }
```

```
}
```

Konštruktor nadtriedy môže byť vyvolaný iba z konštruktora. Volanie konštruktora nadtriedy musí byť uvedené ako prvý príkaz konštruktora.
Nadtrieda môže obsahovať niekoľko konštruktorov. Požadovaný konštruktor nadtriedy sa vyberie podľa typov parametrov uvedených pri jeho volaní.
Ak volanie konštruktora nadtriedy nie je explicitne uvedené, kompilátor automaticky doplní volanie konštruktora nadtriedy bez parametrov. Ak nadtrieda takýto konštruktor neobsahuje, potom vznikne chyba pri kompilácii.
Podobne, ak v triede explicitne nedefinujeme konštruktor, potom jej implicitný konštruktor zavolá konštruktor nadtriedy bez parametrov.

Pozn1: Trieda `Object` obsahuje konštruktor bez parametrov.

Pozn2: Ak je v triede explicitne definovaný konštruktor s parametrami, ale nie je explicitne definovaný konštruktor bez parametrov, tak trieda neobsahuje implicitný konštruktor bez parametrov (kompilátor nedoplní implicitný konštruktor).

Finálne metódy

Finálna metóda je metóda, ktorej implementácia sa v podtriedach nemôže meniť (v podtriede nemôže byť prekrytá). Označuje sa kľúčovým slovom `final`.
Finálne metódy sa napr. používajú pri inicializácii inštančných premenných.
Vo všeobecnosti je vhodné deklarovať metódy volané z konštruktorov ako `final` (prekrytie takejto metódy v podtriede môže mať nežiaduce následky)

príklad:

```
public class Trieda {
    public final void finalnaMetoda(int p1, int p2) {
        //implemntacia metody,
        //ktoru v podtriede nie je mozne prekryt
    }
}
```

Finálne triedy

Finálna trieda je trieda, ktorá nemôže mať podtriedy. Označuje sa kľúčovým slovom `final`.
Finálne triedy je vhodné použiť napr. pri vytváraní tried, ktorých inštancie sú *nemenné* (*immutable*) t.j. ich vnútorný stav (hodnoty atribútov) sa po vytvorení nemení. Potom nemožno vytvoriť podtriedu, ktorá by umožňovala meniť vnútorný stav inštancii. Príkladom je trieda `String`.

príklad:

```
public final class FinalnaTrieda {
    private int atribut1;

    public void metoda() {
        //.....
    }
}
```

Volanie prekryteľnej metódy v konštruktore

Každú zdedenú metódu môžeme prekryť ak túto možnosť explicitne nezakážeme (str. 12). Ak v konštruktore voláme metódu, ktorá môže byť prekrytá v podtriede, môže nastať chyba. Príklad:

```
public class A {
    private int a;

    public A() {
        reset(); //možnosť chyby ak bude metóda prekrytá
    }

    public void reset() {
        a = 1;
        System.out.println("A.reset()");
    }
}
```

```
public class B extends A{
    private int b;

    public B() {
        reset();
    }

    @Override
    public void reset() {
        super.reset();
        b = 1;
        System.out.println("B.reset()");
    }
}
```

Inicializácia inštancie podtriedy neprebehne tak ako sme možno očakávali:

```
public static void main(String[] args) {
    B b = new B();
}
```

výstup:

```
A.reset()
B.reset()
A.reset()
B.reset()
```

Pri inicializácii inštancie sa najprv zavolá konštruktor nadtriedy, ktorý zavolá metódu `reset()`. Táto metóda je však prekrytá a preto sa pri volaní konštruktora nadtriedy vykoná prekrytá verzia z podtriedy. Potom sa v konštruktore podtriedy zavolá znova tá istá metóda `reset()`.

Zabránenie možnosti prekrytia metódy v podtriede nemusí byť vždy vhodné. Iná možnosť opravy je:

```
public class A {
    private int a;

    public A() {
        privateReset();
    }

    private void privateReset() {
        a = 0;
        System.out.println("A.privateReset()");
    }

    public void reset() {
        privateReset();
    }
}
```

```
public class B extends A{
    private int b;

    public B() {
        privateReset();
    }

    private void privateReset() {
        b = 0;
        System.out.println("B.privateReset()");
    }

    @Override
    public void reset() {
        super.reset();
        privateReset();
    }
}
```

Abstraktné metódy a triedy

Abstraktná metóda je metóda deklarovaná bez implementácie. Deklaruje sa kľúčovým slovom `abstract`.

Abstraktná trieda je trieda, z ktorej nemožno vytvoriť inštancie. Je deklarovaná s kľúčovým slovom `abstract`. Môže, ale nemusí obsahovať abstraktné metódy. Ak trieda obsahuje aspoň jednu abstraktnú metódu, potom musí byť deklarovaná s kľúčovým slovom `abstract`.

príklad:
GrafickyPrvok.java:

```
public abstract class GrafickyPrvok {
    private boolean zobrazovat;

    public abstract void kresli();
    public abstract void posun(double posunX, double posunY);
    public boolean jeZobrazeny() {
        return zobrazovat;
    }
}
```

Kruznica.java:

```
public class Kruznica extends GrafickyPrvok{
    private double stredX, stredY;
    private double polomer;

    @Override
    public void kresli() {
        //kod pre vykreslenie
    }

    @Override
    public void posun(double posunX, double posunY) {
        stredX += posunX;
        stredY += posunY;
    }

    public double obvod() {
        return 2 * Math.PI * polomer;
    }
}
```

použitie v metóde main:

```
public static void main(String[] args) {
    //GrafickyPrvok g = new GrafickyPrvok(); <-CHYBA
    Kruznica k = new Kruznica();
}
```

Implementácia rozhrania abstraktnou triedou

Abstraktná trieda nemusí implementovať všetky metódy rozhrania.

príklad:
Rozhrane.java:
```
public interface Rozhranie {
    void metoda1(int parameter1, int parameter2);
    void metoda2(int parameter);
}
```
AbstraktnaTrieda.java:
```
public abstract class AbstraktnaTrieda implements Rozhranie {

    @Override
    public void metoda1 (int parameter1, int parameter2) {
        //implementacia
    }

    //metoda2 zostáva abstraktnou metódou
}
```
KonkretnaTrieda.java:
```
public class KonkretnaTrieda extends AbstraktnaTrieda{

    @Override
    public void metoda2 (int parameter) {
        //implementacia
    }
}
```

Metódy rozhrania sú implicitne abstraktné (ak nie sú default-né alebo statické), takže sa modifikátor `abstract` pri definícii metód rozhrania nemusí používať.
V príklade `metoda2` zostáva v triede `AbstraktnaTrieda` abstraktnou metódou. Je implementovaná až v triede `KonkretnaTrieda`.

Trieda ktorá dedí od nadtriedy aj implemntuje rozhrania

Podľa konvencie sa najprv uvádza klauzula `extends` a potom klauzula `implements`.

príklad:
```
public class Trieda extends Nadtrieda implements Rozhranie1, Rozhranie2{
    //.....
}
```

Viacnásobná dedičnosť

Viacnásobná dedičnosť znamená, že trieda môže priamo dediť od viacerých typov. Ak trieda zdedí z viacerých nadtypov atribúty s rovnakým menom, alebo zdedí implementované metódy s rovnakou signatúrou, nastáva nejednoznačnosť. Jazyky ktoré umožňujú viacnásobnú dedičnosť obsahujú mechanizmy pre riešenie tohto javu. Java neumožňuje viacnásobné dedenie tried, ale umožňuje implementáciu viacerých rozhraní. Ak implementované rozhrania obsahujú abstraktné metódy s rovnakou signatúrou, tak problém s nejednoznačnosťou nie je, pretože trieda ponechá túto metódu ako abstraktnú, alebo ju implementuje. Ak implementované rozhranie obsahujú default-né metódy s rovnakou signatúrou, nastáva nejednoznačnosť, pretože implementácie sa môžu líšiť. V takýchto prípadoch buď kompilátor automaticky rozhodne, ktorá implementácia bude použitá, alebo vznikne chyba pri kompilácii a programátor musí metódu explicitne implementovať.

Dedenie default-ných metód

Abstraktné a default-né metódy rozhraní sú dedené ako inštančné metódy. Ak trieda dedí viac dafault-ných metód s rovnakou signatúrou, java kompilátor sa musí rozhodnúť či a ktorú implementáciu bude trieda dediť. Pri rozhodovaní sa riadi dvoma princípmi:

Inštančné metódy definované v triedach majú prednosť pred default-nými metódami rozhraní.
príklad:

```
public class A {
    public String getString() {
        return "trieda A";
    }
}
```

```
public interface R {
    default String getString() {
        return "rozhranie R";
    }
}
```

```
public class B extends A implements R {
}
```

```
public static void main(String[] args) {
    B b = new B();
    System.out.println(b.getString()); //vytlačí "trieda A"
}
```

Keď nadtypy majú spoločného predka, uplatní sa prekrytá metóda.
Príklad:

```
public interface IA {
    default String getString() {
        return "rozhranie IA";
    }
}
```

```
public interface IB extends IA{
    @Override
    default String getString() {
        return "rozhranie IB";
    }
}
```

```
public interface IC extends IA{
}
```

```
public class T implements IB, IC{
}
```

```
public static void main(String[] args) {
    T objekt = new T();
    System.out.println(objekt.getString()); //vytlačí "rozhranie IB"
}
```

Ak majú rovnakú signatúru nezávisle definované default-né metódy, alebo defaultné metódy s abstraktnými metódami, tak vznikne chyba pri kompilácii. Metódu je potrebné explicitne prekryť.
Príklad:

```
public interface IA {
    default String getString() {
        return "rozhranie IA";
    }
}
```

```
public interface IB {
    default String getString() {
        return "rozhranie IB";
    }
}
```

```
public class Trieda implements IA, IB {
    @Override
    public String getString() {
        return "trieda T"
                + ", "
                + IA.super.getString()   //volanie default metódy z rozhrania IA
                + ", "
                + IB.super.getString(); //volanie default metódy z rozhrania IB
    }
}
```

Kedy je vhodné použiť dedičnosť

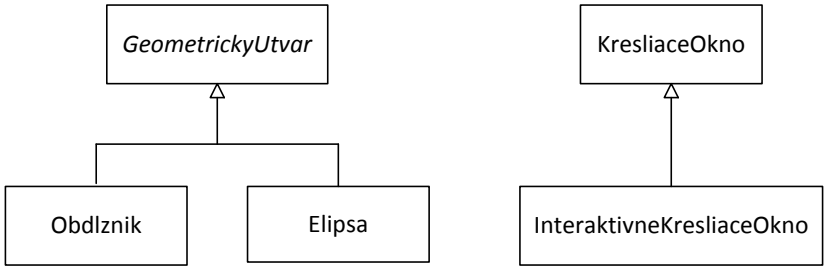
Dedičnosť vyjadruje vzťah: zovšeobecnenie (nadtrieda) – špecializácia (podtrieda)

Dedenie je vhodné používať tam, keď sú inštancie potomka špeciálnym druhom predka.

Dedičnosť nie je vhodné použiť len preto, aby sme využili kód definovaný v inej triede (v takom prípade je lepšie použiť iný druh asociácie napr. kompozíciu), napr.:

- obdĺžnik nie je podtriedou bodu, bod je atribút obdĺžnika,
- študent nie je potomkom dátumu nástupu (čo ak by sme chceli zdediť ešte raz dátum, aby sme ho mohli použiť ako dátumu ukončenia štúdia, zápisu skúšky a pod.)
- úsečka definovaná začiatočným a koncovým bodom nie je potomkom bodu, ale obsahuje dva atribúty triedy bod

Príklady vhodného použitia dedenia:



Ak nadtrieda obsahuje iba spoločnú abstrakciu a nemá zmysel vytvárať jej inštancie, je vhodné deklarovať triedu ako abstraktnú. Trieda `GeometrickyUtvar` je abstraktná (označenie šikmým písmom). Trieda `KresliaceOkno` nie je abstraktná, pretože jej inštancie využijeme pre samotné kreslenie. Trieda `InteraktivneKresliaceOkno` rozširuje nadtriedu o možnosť interakcie pomocou myši.

Pri zisťovaní či je dedičnosť vhodná, často pomôže pravidlo „je“ (angl „is a“).
Príklady:

- obdĺžnik je geometrickým útvarom
- elipsa nie je bodom
- elipsa nie je stredom
- elipsa je geometrickým útvarom
- interaktívne kresliace okno je kresliacim oknom
- disk je tovarom
- študent nie je dátumom
- študent je osobou

Liskovej princíp substitúcie (Liskov Substitution Principle)

Kde možno použiť inštanciu predka, tam musí byť možné použiť inštanciu potomka.

Nech $q(a)$ je preukázateľná vlastnosť objektu a typu A. Potom $q(b)$ by mala platiť pre objekt b typu B, kde B je podtyp typu A.

Príklad:
Všade tam kde je možné použiť inštanciu triedy `KresliaceOkno`, je možné použiť aj inštanciu `InteraktívneKresliaceOkno`.

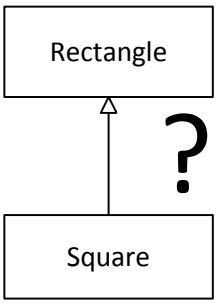
V podtriede musia byť:
- vstupné podmienky nezmenené, alebo zoslabené
- výstupné podmienky nezmenené, alebo zosilnené

Vstupné podmienky – podmienky, ktoré musia spĺňať vstupné údaje
Výstupné podmienky – podmienky, ktoré musia byť splnené na výstupe. Vyjadrujú vzťah medzi vstupnými údajmi, stavom pred vykonaním a výstupnými údajmi a stavom po vykonaní.

Príklad zoslabenia a zosilnenia vstupných podmienok prekrytej metódy
pôvodné (v nadtriede):
vstup musí byť v intervale <0,10>
zoslabené (v podtriede):
vstup musí byť v intervale <0, 20>. Metóda v podtriede má väčší rozsah vstupných hodnôt, čo nebráni jej použitiu všade tam, kde mohla byť použitá metóda nadtriedy
zosilnené (v podtriede):
vstup musí byť v intervale <0,5>. Metóda v podtriede má menší rozsah vstupných hodnôt, nemožno ju použiť všade tam, kde bolo možné použiť metódu nadtriedy

Príklad zosilnenia a zoslabenia výstupných podmienok prekrytej metódy.
pôvodné (v nadtriede):
výstupná hodnota je v rozsahu <0,10>
zosilnené (v podtriede):
výstupná hodnota je v rozsahu <0,5>. Metóda vracia hodnoty v menšom rozsahu, čo nebráni jej použitiu tam kde môže byť výstupná hodnota v rozsahu <0,10>
zoslabené (v podtriede):
výstupná hodnota je v rozsahu <0, 20>. Na mieste, kde predpokladáme výstupné hodnoty v interval <0,10> nie je splnený tento predpoklad

Príklad:
Z matematického hľadiska je štvorec špeciálnym druhom obdĺžnika.
Je vhodné dediť štvorec od obdĺžnika?



Rectangle.java

```
/*
 * invariant (podmienka ktora plati vzdy):
 * width > 0 && height > 0
 */
public class Rectangle {
    private int width;
    private int height;

    /*
     * post:
     *     sirka a vyska nastavena na 1
     */
    public Rectangle() {
        this.width = 1;
        this.height = 1;
    }

    /*
     * pre (preconditions = vstupne podmienky):
     *     width > 0
     * post (postconditions = vystupne podmientky):
     *     (nova this.width == width)
     *     &&
     *     (hodnoty ostatnych atributov sa nemenia)
     */
    public void setWidth(int width) {
        assert width > 0 : "chybne nastavenie sirky";

        this.width = width;
    }

    /*
     * pre:
     *     height > 0
     * post:
     *     (nova this.height == height)
     *     &&
     *     (hodnoty ostatnych atributov sa nemenia)
     */
    public void setHeight(int height) {
        assert height>0 : "chybne nastavenie vysky";

        this.height = height;
    }

    /*
     * post:
     *     navratova hodnota je sirka obdlznika && stav objektu sa nemeni
     */
    public int getWidth() {
        return width;
    }

    /*
     * post:
     *    navratova hodnota je vyska obdlznika && stav objektu sa nemeni
     */
    public int getHeight() {
        return height;
    }
```

```
}
```

Square.java

```
/*
 * invariant (podmienka ktora plati vzdy):
 * sirka == vyska && sirka > 0
 */
public class Square extends Rectangle {

    /*
     * nedodrzana vystupna podmienka z nadtriedy
     */
    @Override
    public void setWidth(int width) {
        super.setWidth(width);
        super.setHeight(width);
    }

    /*
     * nedodrzana vystupna podmienka z nadtriedy
     */
    @Override
    public void setHeight(int height) {
        super.setHeight(height);
        super.setWidth(height);
    }
}
```

MainRectangleSquare.java

```
public class MainRectangleSquare {
    public static void main(String[] args) {
      //Rectangle rect = new Rectangle();
        Rectangle rect = new Square();

        setDoubleWidth(rect);

        System.out.println("uspesny koniec programu");
    }

    //Metoda nastavi sirku obdlznika tak,
    //aby bola dvakrat vacsia nez je vyska
    private static void setDoubleWidth(Rectangle r) {
        int newWidth = 2 * r.getHeight();

        r.setWidth(newWidth);

        assert r.getWidth() == 2 *r.getHeight()
               : "nepodarilo sa nastavit obdlznik";
    }
}
```

Vhodnosť dedičnosti môže závisieť od vonkajších podmienok. Metóda `setDoubleWidth()` nebude fungovať správne ak jej argument je typu `Square`. Je to spôsobené nedodržaním výstupných podmienok zdedenej metódy `setWidth()`. Súčasťou výstupnej podmienky metódy `setWdith()` v triede `Rectangle` je nezmenenie výšky, táto časť výstupnej podmienky ale nie je dodržaná v po prekrytí v triede `Square`.