Prednáška 8: Polymorfizmus. Abstraktné triedy

Ján Lang

kanc. 4.34, lang@fiit.stuba.sk, http://www2.fiit.stuba.sk/~lang/zoop/

Ústav informatiky, informačných systémov a softvérového inžinierstva

Fakulta informatiky a informačných technológií

Slovenská technická univerzita v Bratislave

21. november 2023

Menný priestor

- Prvá časť názvu balíčka je reverzná forma názvu internetovej domény tvorcu (dve rovnomenné triedy v jednom, dvoch balíkoch?)

```
package sk.stuba.fiit.myClasses;
    public class Vector {
        public Vector() {
        System.out.println("sk.stuba.fiit.myClasses.Vector"
          );
    }
}

package sk.stuba.fiit.myClasses;
public class List {
    public List() {
    System.out.println("sk.stuba.fiit.myClasses.List");
    }
}
```

Menný priestor

- Prvá časť názvu balíčka je reverzná forma názvu internetovej domény tvorcu (dve rovnomenné triedy v jednom, dvoch balíkoch?)

```
package sk.stuba.fiit.myClasses;
    public class Vector {
        public Vector() {
        System.out.println("sk.stuba.fiit.myClasses.Vector"
          );
    }
}

package sk.stuba.fiit.myClasses;
public class List {
    public List() {
    System.out.println("sk.stuba.fiit.myClasses.List");
    }
}
```

Menný priestor

- Prvá časť názvu balíčka je reverzná forma názvu internetovej domény tvorcu (dve rovnomenné triedy v jednom, dvoch balíkoch?)

```
import java.util.*;

public class LibTest {
   public static void main(String[] args) {
        Vector vec = new Vector();
        List lis = new List();
   }
}
```

Pracovný priestor

- Eclipse -> File -> Switch Workspace -> Other...
- C:\Users\lang\workspace

- Oba súbory umiestnené v konkrétnom podadresári súborového systému
- ...workspace**nazovProjektu**\src**sk\stuba\fiit\myClasses**
  - **sk\stuba\fiit\myClasses** - pomenovanie balíka
  - C:\Users\lang\workspace – prvá časť v premennej prostredia

- Vo win Control Panel\System and Security\System – Advanced system settings – Environment variables, ALEBO Start Search - "env- "Edit the system environment variables"

  CLASSPATH=.;D:\JAVA\LIB; C:\Users\lang\workspace

V prípade .jar je potrebné uviesť absolútnu cestu

Pracovný priestor

- Ako náhle premenná CLASSPATH je správne nastavená, nasledovný súbor môže byť umiestnený v ľubovoľnom adresári

```
import    sk.stuba.fiit.myClasses.*;
   //     sk\stuba\fiit\myClasses.*;
public class LibTest {
   public static void main(String[] args) {
      Vector v = new Vector();
      List l = new List();
   }
}
```

- Keď kompilátor narazí na príkaz import pre našu knižnicu myClasses začne prehľadávať v adresároch špecifikovaných v CLASSPATH, hľadá podadresár **sk\stuba\fiit\myClasses**, potom hľadá kompilované súbory príslušných tried (Vector.class a List.class). Obe triedy a požadované metódy tried musia byť verejné

Pracovný priestor

- Prvá časť názvu balíčka je reverzná forma názvu internetovej domény tvorcu (dve rovnomenné triedy v jednom, dvoch balíkoch?)

```
import sk.stuba.fiit.myClasses.*;
import java.util.*;

public class LibTest {
   public static void main(String[] args) {
        sk.stuba.fiit.myClasses.Vector v = new
   sk.stuba.fiit.myClasses.Vector();
        Vector vec = new Vector();
        List l = new List();
   }
}
```

Menný priestor

- Kolízie. Dva importované balíky/knižnice obsahujú rovnomenné triedu/triedy

```
import sk.stuba.fiit.myClasses.*;
import java.util.*;
```

- `java.util.*` obsahuje triedu `Vector`
- Problém nastane ak chcem vytvoriť inštanciu triedy `Vector`

```
Vector v = new Vector();
```

- Otázne je na ktorú triedu sa odkazujem... Z uvedeného to nevie ani kompilátor ani čitateľ.
- Ak chcem použiť štandardný `Vector`, musím uviesť:

```
java.util.Vector v = new java.util.Vector();
```

- Ak chcem použiť vlastný Vector, musím uviesť:

```
sk.stuba.fiit.myClasses.Vector v = new
sk.stuba.fiit.myClasses.Vector();
```

Vlastné knižnice

- Toto poznanie umožňuje vytváranie vlastných knižníc napr. na redukciu resp. eliminovanie duplicity v kóde

```
public class P {
    public static void rint(String s) {
      System.out.print(s);
    }
    public static void rintln(String s) {
      System.out.println(s);
    }
}
```

```
public class ToolTest {
    public static void main(String[] args) {
        P.rintln("Available from now on!");
        P.rintln("" + 100); // Force it to be a String
        P.rintln("" + 100L);
        P.rintln("" + 3.14159);
    }
}
```

Príklad

```
class Bart extends Homer {
    void doh(Milhouse m) {
        System.out.println("doh(Milhouse)");
    }
}
```

- Trieda nadtypu disponuje viacnásobne preťaženou metódou. Preťaženie tejto metódy v triede podtypu neschová žiadnu z verzii metód nadtypu...

```
class Homer {
    char doh(char c) {
        System.out.println("doh(char)");
        return 'd';
    }

    float doh(float f) {
        System.out.println("doh(float)");
        return 1.0f;
    }
}
```

```
class Instrument {
	public void play() {}

	static void tune(Instrument i) {
	i.play();
	}
}
```

Upcasting

- Veľmi dôležitý aspekt dedenia
- Vzťah medzi triedou nadtypu a triedou podtypu
- Vzťah vyjadrený ako: „ nová trieda je typu existujúcej triedy"

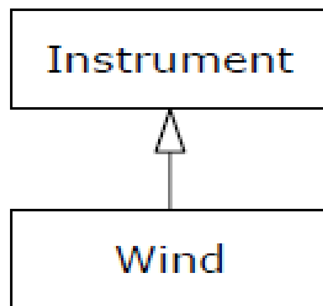
```
// Objekty triedy Wind sú Inštrumenty
// pretože majú to isté rozhranie:
public class Wind extends Instrument {
	public static void main(String[] args)
	{

	Wind flute = new Wind();
	Instrument.tune(flute); // Upcasting
	}
}
```

- Metóda `tune()` akceptuje referenciu typu `Instrument`
- Napriek tomu vo `Wind.main()` je `tune()` metóda volaná s parametrom `Wind` referencie
- Upcasting-om v tomto prípade nazývame „konverziu" **Wind** referencie na **Instrument** referenciu

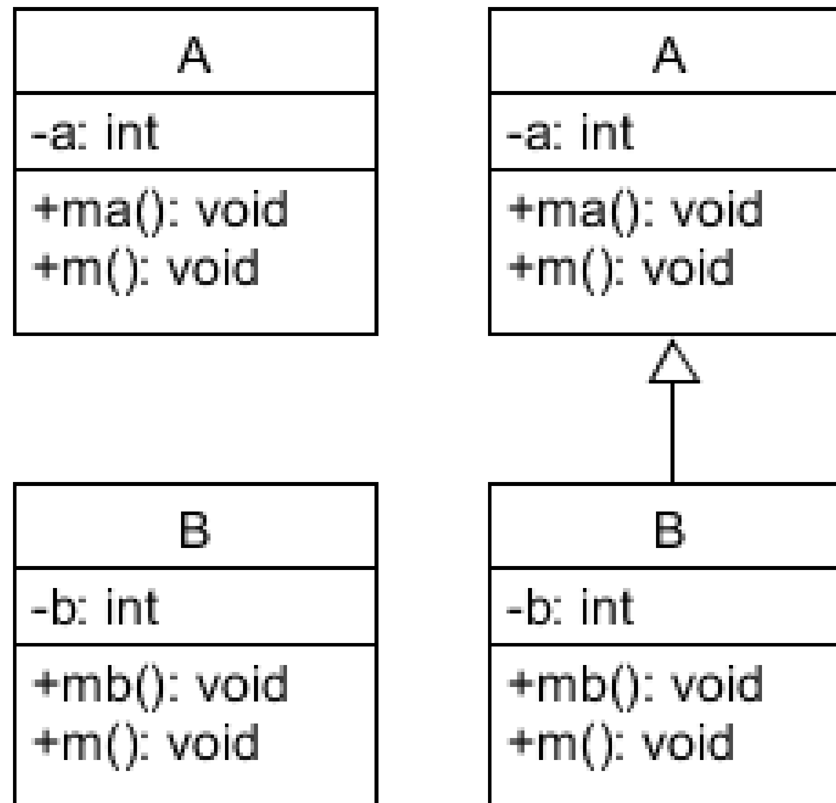
Upcasting

- „pretypovanie" z odvodeného typu na bázový typ je v UML diagramoch tried zasadené do vzťahu dedenia. Vzťah dedenia je principiálne znázorňovaný zdola nahor (ale nie je to pravidlo), preto upcasting



- Upcasting je bezpečný na rozdiel od Downcastingu. Podtyp **môže** obsahovať viac metód než jeho nadtyp, ale **musí** obsahovať aspoň metódy nadtypu

Upcasting



Skladanie vs. dedenie

- Skladanie a dedenie umožňuje znovupoužitie existujúceho kódu v rámci vytváraného nového, inštanciáciu existujúcich typov vo vnútri vytváraného nového typu
- Skladanie explicitné vyjadrenie, dedenie implicitné vyjadrenie
- Skladanie vo všeobecnosti keď požadujem vlastnosti existujúcej triedy (nie jej rozhranie) vo vnútri novej triedy. Používateľ vníma rozhranie novej triedy než rozhranie vložených objektov. Preto je toto vhodné definovať ako `private`. Vzťah vyjadrený `has-a`, „Má“

Skladanie vs. dedenie

- Dedenie sprístupňuje inerface nadtypu na jeho reimplementáciu prípadne rozšírenie. Vzťah vyjadrený `is-a`, „Je“
- Skladanie vs. dedenie – dá odpoveď na otázku: „...budeme niekedy potrebovať realizovať upcasting?“
- Výhodou dedenia – podporuje inkrementálny vývoj. Nový program nespôsobuje problémy v už existujúcom kóde. Toto ohraničuje chyby v priestore novovytvoreného kódu
- Integračné testovanie

Preťaženie vs. Prekonávanie

V rámci jednej triedy

- V prípade, že sa rovnomenné metódy triedy líšia v parametroch dochádza k preťaženiu

```
Class NejakaTrieda {
Obyvatel(String meno, String priezvisko, int vek) { ... }
Obyvatel(Obyvatel o) { ... }
Obyvatel() { ... }
}
```

Preťaženie vs. Prekonávanie

V rámci jednej triedy

- V prípade, že sa rovnomenné metódy triedy nelíšia v parametroch dochádza k duplicite a kompilátor hlási chybu!!!

```
Class NejakaTrieda {
Obyvatel(String meno, String priezvisko, int vek) { ... }
Obyvatel(String meno, String priezvisko, int vek) { ... }
Obyvatel(Obyvatel o) { ... }
Obyvatel() { ... }
}
```

Preťaženie vs. Prekonávanie

V rámci hierarchie dedenia

- V prípade, že sa rovnomenné metódy (zdedené z Nadtypu a novo pridané do Podtypu) líšia v parametroch dochádza k preťaženiu

```
class Nadtyp {
Obyvatel(String meno, String priezvisko, int vek) { ... }
}

class Podtyp {
Obyvatel(Obyvatel o) { ... }
Obyvatel() { ... }
}
```

Preťaženie vs. Prekonávanie

V rámci hierarchie dedenia

- V prípade, že sa rovnomenné metódy (zdedené z Nadtypu a novo pridané do Podtypu) nelíšia v parametroch dochádza k prekonaniu. Deklarácia nestatickej metódy rovnakej signatúry v Podtype prekonáva (overrides) pôvodnú metódu Nadtypu

```
class Nadtyp {
mObyvatel(String meno, String priezvisko, int vek) { ...
   }
}

class Podtyp {
mObyvatel(String meno, String priezvisko, int vek) { ...
   }
}
```

Riadenie prekonávania

- **Zakázať** / Povoliť / Prikázať
- Ako???

Riadenie prekonávania

- **Zakázať** / Povoliť / Prikázať
- V prípade, že existuje dôvod prečo nedovoliť prekonávanie niektorej z metód nadtypu je potrebné takúto metódu označiť kľúčovým slovom `final`. Ak je v rodičovskej triede metóda označená ako `final` môžeme ju v zdedenej triede preťažiť ale nie prekonať

```
final String vypis() { return ("nemozno ma prekonat"); }
```

Riadenie prekonávania

- Zakázať / **Povoliť** / Prikázať
- Ako???

Riadenie prekonávania

- Zakázať / Povoliť / **Prikázať**
- Ako???

Riadenie prekonávania

- Zakázať / Povoliť / **Prikázať**
- V prípade, že existuje dôvod prečo prikázať prekonávanie niektorej z metód nadtypu je potrebné takúto metódu označiť kľúčovým slovom `abstract` alebo použiť `interface` (rozhrania nabudúce v prednáške)
- Spôsob akým je možné vynútenie implementácie metódy rovnakej signatúry v zdedenej triede.

```
public abstract class Potraviny {
   abstract void doplnVitaminy(Obyvatel o);
}
```

- Týmto je zaručené že každá potravina bude vedieť svojim spôsobom doplniť vitamíny obyvateľovi, ktorý ju požíva. Každá potravina t.j. trieda, ktorá bude dediť od triedy `Potraviny`

Riadenie prekonávania

- Zakázať / Povoliť / **Prikázať**
- V prípade, že existuje dôvod prečo prikázať prekonávanie niektorej z metód nadtypu je potrebné takúto metódu označiť kľúčovým slovom `abstract` alebo použiť `interface` (rozhrania nabudúce v prednáške)
- Spôsob akým je možné vynútenie implementácie metódy rovnakej signatúry v zdedenej triede.

```
public abstract class Potraviny {
   abstract void doplnVitaminy(Obyvatel o);
}
```

- Týmto je zaručené že každá potravina bude vedieť svojim spôsobom doplniť vitamíny obyvateľovi, ktorý ju požíva. Každá potravina t.j. trieda, ktorá bude dediť od triedy `Potraviny`

Riadenie prekonávania

```
public class Mrkva extends Potraviny {
    void doplnVitaminy(Obyvatel o) {
    o.silaZraku += 10;
    }
}
public class Paprika extends Potraviny {
    void doplnVitaminy(Obyvatel o) {
    o.silaZraku += 1;
    }
}
public class Metanol extends Potraviny {
    void doplnVitaminy(Obyvatel o) {
    o.silaZraku -= 10;
    }
}
```

Abstraktná trieda

- Kľúčové slovo `abstrakt` nasleduje návratový typ, názov metódy, formálne parametre v našom prípade objekt triedy `Obyvatel` bez uvedenia tela metódy, ale s uvedením bodkočiarky

```
abstract void doplnVitaminy(Obyvatel o);
```

- Abstraktné triedy sa využívajú na definovanie spoločných **charakterových** rysov podtried. Reprezentujú akési šablóny pre vytváranie konkrétnych podtried
- Môžu obsahovať vlastnosti a metódy
- Z abstraktnej triedy sa nedá vytvoriť inštancia (objekt)
- Generický typ pre nové typy, ktoré ho budú rozširovať

Abstraktná trieda

- Abstraktná trieda môže taktiež obsahovať metódy bez implementácie – tzv. abstraktne metódy
- Abstraktná trieda nemusí obsahovať len abstraktne metódy. Ale abstraktná metóda môže byť len v abstraktnej triede

```
public abstract class Potraviny {
    abstract void doplnVitaminy(Obyvatel o);
    public String toString() {
      return "Nazov potraviny: ";
    }
}
```

- V zdedenej triede sa musia implementovať len abstraktné metódy abstraktnej triedy, ostatné metódy je možné ponechať bezo zmeny, preťažiť alebo prekonať pokiaľ nie sú označené ako `final`

Abstraktná trieda

- Trieda Shape je abstrakná
- Obsahuje 2 abstraktné metódy: draw a resize
- Metóda moveTo nie je abstraktná (má svoju implementáciu)

```
abstract class Shape {
    int x, y;
    void moveTo(int x, int y) {
        this.x = x;
        this.y = y;
    }
    abstract void draw();
    abstract void resize();
}
```

Príklad

- ...TIJ c07:Shapes.java

Polymorfizmus

- Polymorfizmus je mechanizmus, ktorý umožňuje objektom rôznych typov odpovedať na volanie rovnakej metódy rôznym spôsobom.
- Polymorfizmus (viacznačnosť) je schopnosť objektu nadobúdať viacero foriem. Najčastejšie použitie polymorfizmu je vtedy, keď v referencii na triedu rodiča používame odkaz na objekty triedy potomka.

```
public class Zviera{ }
public class Kon extends Zviera
```

Polymorfizmus

- Polymorfizmus je mechanizmus, ktorý umožňuje objektom rôznych typov odpovedať na volanie rovnakej metódy rôznym spôsobom.
- Polymorfizmus (viacznačnosť) je schopnosť objektu nadobúdať viacero foriem. Najčastejšie použitie polymorfizmu je vtedy, keď **v referencii na triedu rodiča používame odkaz na objekty triedy potomka**.

```
public class Zviera{ }
public class Kon extends Zviera
```

Polymorfizmus

```
public class Zviera{ }
public class Kon extends Zviera
```

Polymorfizmus

```
public class Zviera{ }
public class Kon extends Zviera

Kon j = new Kon();
Zviera z = j;
Object o = j;
```

Polymorfizmus

```
public class Zviera{ }
public class Kon extends Zviera

Kon j = new Kon();
Zviera z = j;
Object o = j;
```

- Čo môžeme tvrdiť o triede Kôň?

Polymorfizmus

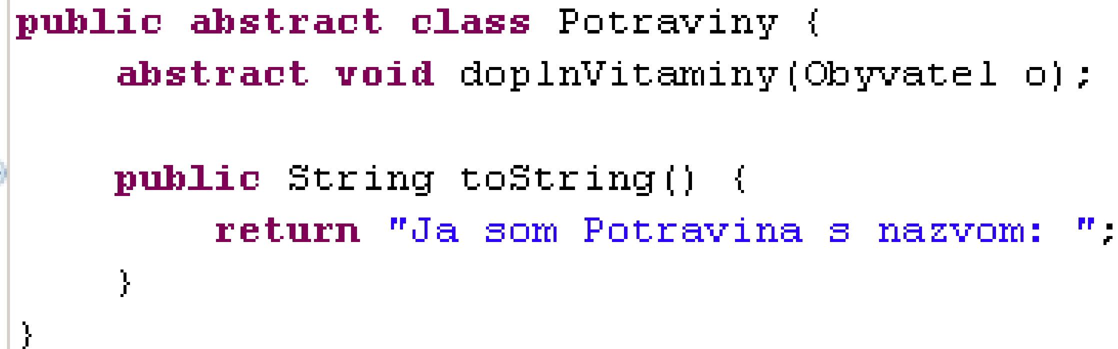
```
public class Zviera{ }
public class Kon extends Zviera

Kon j = new Kon();
Zviera z = j;
Object o = j;
```

- O triede Kôň môžeme tvrdiť:
  - Kôň je *Kon*
  - Kôň je zviera (Zviera)
  - Kôň je objekt (Object)

- Všetky referenčné premenné (`j,z,o`) odkazujú na jeden a ten istý objekt - inštanciu triedy `Kon`

Polymorfizmus



- Metóda prekonáva `java.lang.Object.toString`
- Rôzne implementácie tejto metódy volané uniformne vyústia rôznorodým prejavom

Polymorfizmus

```
public class Mrkva extends Potraviny {
    void doplnVitaminy(Obyvatel o) {
        o.silaZraku += 100;
    }
    public String toString() {
        return super.toString() + "Mrkva";
    }
}
```

- Metóda prekonáva predpísanú metódu abstraktnej triedy
- Implementácia metódy v podtype je špecifická

Polymorfizmus

```
public class Paprika extends Potraviny {
    private double vitaminA = 10;

    void doplnVitaminy(Obyvatel o) {
        o.silaZraku += vitaminA;
    }

    public String toString() {
        return super.toString() + "Paprika";
    }
}
```

```
public class Metanol extends Potraviny{
    void doplnVitaminy(Obyvatel o) {
        o.silaZraku -= 1000;
    }
    public String toString() {
        return super.toString() + "Metanol";
    }
}
```

Polymorfizmus

```
Potraviny[] nakupnyKosik = new Potraviny[5];
nakupnyKosik[0] = new Paprika();
nakupnyKosik[1] = new Mrkva();
nakupnyKosik[2] = new Metanol();
nakupnyKosik[3] = new Paprika();
nakupnyKosik[4] = new Paprika();

for (Potraviny p : nakupnyKosik) {
    System.out.println(p);
}
```

- Otázka udržiavania inštancií tried
- `nakupnyKosik` referencuje pole potravín. Každý prvok poľa je referenciou, ktorá je inicializovaná inštanciou podtypu triedy `Potraviny`
- ...v referencii typu rodiča používame odkaz na objekty triedy potomka - Upcasting

Polymorfizmus

```
Potraviny[] nakupnyKosik = new Potraviny[5];
nakupnyKosik[0] = new Paprika();
nakupnyKosik[1] = new Mrkva();
nakupnyKosik[2] = new Metanol();
nakupnyKosik[3] = new Paprika();
nakupnyKosik[4] = new Paprika();

for (Potraviny p : nakupnyKosik) {
    System.out.println(p);
}
```

- Volanie metódy `toString`.

```
Ja som Potravina s nazvom: Paprika
Ja som Potravina s nazvom: Paprika
Ja som Potravina s nazvom: Mrkva
Ja som Potravina s nazvom: Metanol
Ja som Potravina s nazvom: Paprika
Ja som Potravina s nazvom: Paprika
```

Polymorfizmus

```
public class Obyvatel extends Clovek {

        Auto sukromneAuto;
        int vek;
        int silaZraku = 100;
```

```
Obyvatel o = new Obyvatel("Jano", "Hladny", 22);
System.out.println(o.silaZraku);
```

```
Do nasej hry sme pridali obyvatela s menom: Jano, priezviskom: Hladny a vekom: 22
```

```
for (Potraviny p : nakupnyKosik) {
    p.doplnVitaminy(o);
    System.out.println(o.getSilaZraku());
}
```

Polymorfizmus

```
for (Potraviny p : nakupnyKosik) {
    p.doplnVitaminy(o);
    System.out.println(o.getSilaZraku());
}
```

KEDY sa rozhodne o tom, ktorá verzia metódy sa spustí?

Polymorfizmus

```
for (Potraviny p : nakupnyKosik) {
    p.doplnVitaminy(o);
    System.out.println(o.getSilaZraku());
}
```

KEDY sa rozhodne o tom, ktorá verzia metódy sa spustí?

- Počas behu programu
- Po kompilácii

Polymorfizmus

```
for (Potraviny p : nakupnyKosik) {
    p.doplnVitaminy(o);
    System.out.println(o.getSilaZraku());
}
```

O tom, ktorá verzia metódy sa spustí sa rozhodne až pri zavolaní metódy v bežiacom programe. Teda nie pri kompilácii ale počas behu programu.

Polymorfizmus

```
public class Obyvatel extends Clovek {

        Auto sukromneAuto;
        int vek;
        int silaZraku = 100;
```

```
Obyvatel o = new Obyvatel("Jano", "Hladny", 22);
System.out.println(o.silaZraku);
```

```
Do nasej hry sme pridali obyvatela s menom: Jano, priezviskom: Hladny a vekom: 22
```

```
for (Potraviny p : nakupnyKosik) {
    p.doplnVitaminy(o);
    System.out.println(o.getSilaZraku());
}
```

```
100
110
210
-790
-780
-770
```

- Uplatnenie polymorfizmu
- Dopad na stav objektu triedy `Obyvatel`

Upcasting, Downcasting

- Dynamické pretypovanie - smerom hore (upcasting) a pretypovanie smerom dole (downcasting)
- Majú zmysel len pri použití dedenia
- Java umožňuje objektu vytvoreného z podtriedy byť považovaný za objekt rodičovskej triedy. Táto vlastnosť sa nazýva upcasting. Upcasting sa vykonáva automaticky, zatiaľ čo downcasting musí byť vykonaný explicitne na to určeným príkazom
- Upcasting a downcasting nie je pretypovanie ako ho chápeme pri primitívnych dátových typoch. Pretypovanie nemení typ objektu! Iba ho označí iným spôsobom

Upcasting, Downcasting

```java
public class Clovek {

    String meno, priezvisko;
    Datum datumNarodenia;

    Clovek() {
    }
}
```

```java
public class Obyvatel extends Clovek {

        Auto sukromneAuto;
        int vek;
        int silaZraku;
        int rozpocet;

        void zaplatDanZNehnutelnosti(int dan) {
            rozpocet-=dan;
        }
}
```

```java
Obyvatel o = new Obyvatel();
System.out.println(o);
o.zaplatDanZNehnutelnosti(100);
Clovek c = o; // upcasting
System.out.println(c);
// c.zaplatDanZNehnutelnosti(100);  CHYBA!!!;
```

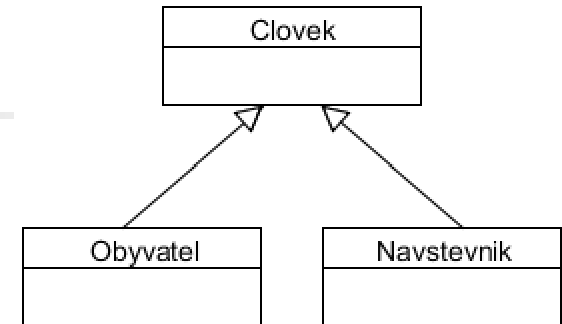
- Vytvorili sme objekt typu `Obyvatel`. Necháme ho zaplatiť daň z nehnuteľnosti. Ďalej vytvoríme referenčnú premennú `c` typu `Clovek`. Referencii `c` priradíme referenciu `o` (ide o upcasting)

Upcasting, Downcasting

```
Obyvatel o = new Obyvatel();
System.out.println(o);
o.zaplatDanZNehnutelnosti(100);
Clovek c = o; // upcasting
System.out.println(c);
// c.zaplatDanZNehnutelnosti(100);  CHYBA!!!;
```

- Referenčná premenná `c` je definovaná ako `Clovek`
- Priradením `c = o;` budeme na `Obyvatela` pozerať ako na `Cloveka`
- `c` je vlastne typu `Clovek`, ale v tomto stave sú skryté všetky špecifické metódy triedy `Obyvatel`
- Inými slovami, objekt referencovaný `c` je `Obyvatel`, ale správa sa ako `Clovek` (neobsahuje špecifické metódy triedy `Obyvatel`)

Upcasting, Downcasting



- Na obrázku je v hierarchii tried trieda `Obyvatel` a `Navstevnik` na jednej úrovni. To znamená, že medzi sebou nemajú žiaden vzťah. Preto nemôže byť objekt `Obyvatel` pretypovaný na objekt `Navstevnik` (a opačne)
- O objekte typu `Obyvatel` (`Obyvatel o = new Obyvatel ();`) môžeme trvdiť:
  - Kompilátor Java pracuje s objektom `o` ako s inštanciou triedy `Object`
  - Objekt `o` je typu `Obyvatel`; má k dispozícii všetky metódy z triedy `Clovek` aj `Obyvatel`
  - Objekt `o` sa dá pretypovať na objekt `Clovek`
  - Objekt `o` sa nedá pretypovať na objekt `Navstevnik`

Downcasting

- Na rozdiel od pretypovania smerom hore, pretypovanie smerom dole musí byť vždy určené explicitne

```
Obyvatel o1 = new Obyvatel();
// upcasting - implicitné pretypovanie smerom "hore" na Cloveka
Clovek c = o1;
// downcasting - explicitné pretypovanie smerom "dole" na Obyvatela
Obyvatel o2 = (Obyvatel) c;
```

- Pri samotnom pretypovaní smerom hore nemôže nikdy nastať neúspech. Teda, upcasting sa vždy podarí
- Pri pretypovaní smerom dole hrozí problém. Ale ak máme skupinu rôznych `Clovekov` a každého chceme pretypovať na `Obyvatela`, je viac ako pravdepodobné, že niektorí budú napríklad inštancie triedy `Navstevnik` a vtedy pretypovanie stroskotá na nekompatibilite typov a vygeneruje sa výnimka `ClassCastException`

Downcasting

- Príklad "bezpečného" pretypovania smerom dolu. Pomocou operátora *instanceof*, zistíme, či je objekt daného typu...

```
Obyvatel o1 = new Obyvatel();
Navstevnik n1 = new Navstevnik();
Clovek c1 = o1;              //upcasting na Clovek
Clovek c2 = n1;              //upcasting na Clovek
Obyvatel o2 = null;
Navstevnik n2 = null;
if (c1 instanceof Obyvatel) {
    o2 = (Obyvatel) c1;      //downcasting na Obyvatel
}
if (c2 instanceof Navstevnik) {
    n2 = (Navstevnik) c2;    //downcasting na Navstevnik
}

System.out.println(o2);
System.out.println(n2);
```

Downcasting

- Pretypovanie sa nie vždy dá robiť oboma smermi. Vytvorme objekt `c` typu `Clovek`

```
Clovek c = new Clovek();
Obyvatel o = (Obyvatel) c;
System.out.println(o);
Navstevnik n = (Navstevnik) c;
System.out.println(n);
```

- Tento objekt sa nedá pretypovať ani na objekt `Obyvatel`, ani na objekt `Navstevnik`. Pozoruhodné je, že uvedený program je bez syntaktických chýb. Dá sa skompilovať. Program padne až po spustení, pri spracovávaní

```
Obyvatel o = (Obyvatel) c;
```

- Tu je veľmi dobre viditeľná vlastnosť polymorfizmu, keď sa vyberá ktorá z metód sa spustí. V tomto prípade sa o výbere správnej metódy nerozhoduje pri kompilácii ale až pri samotnom spustení programu. Po spustení programu dostaneme chybové hlásenie

Downcasting

- Pretypovanie sa nie vždy dá robiť oboma smermi. Vytvorme objekt `c` typu `Clovek`

```
Clovek c = new Clovek();
Obyvatel o = (Obyvatel) c;
System.out.println(o);
Navstevnik n = (Navstevnik) c;
System.out.println(n);
```

- Tento objekt sa nedá pretypovať ani na objekt `Obyvatel`, ani na objekt `Navstevnik`. Pozoruhodné je, že uvedený program je bez syntaktických chýb. Dá sa skompilovať. Program padne až po spustení, pri spracovávaní

```
Obyvatel o = (Obyvatel) c;
```

- Tu je veľmi dobre viditeľná vlastnosť polymorfizmu, keď sa vyberá ktorá z metód sa spustí. V tomto prípade sa o výbere správnej metódy nerozhoduje pri kompilácii, ale až počas samotného behu programu. Po spustení programu dostaneme chybové hlásenie

TODO nezabudnite

- Aj vy môžete pomôcť vylepšiť tento predmet študentom pre nasledujúci akademický rok. Vaše odporúčanie, komentár či otázka.

  ...cez spätnoväzobný formulár.