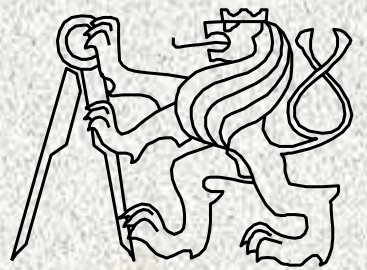
# Třídy, polymorfismus



A0B36PR2-Programování 2

Fakulta elektrotechnická
České vysoké učení technické

# Obsah přednášky

- Na jednoduchém příkladu si zopakujeme pojmy
  - Zapouzdření, opakování
  - dědičnost, opakování
  - Polymorfismus
    - Abstraktní třída
    - Interface
  - Výčtové typy
  - Modelování (pro zájemce)

# Zapouzdření, settery, gettery, accessory

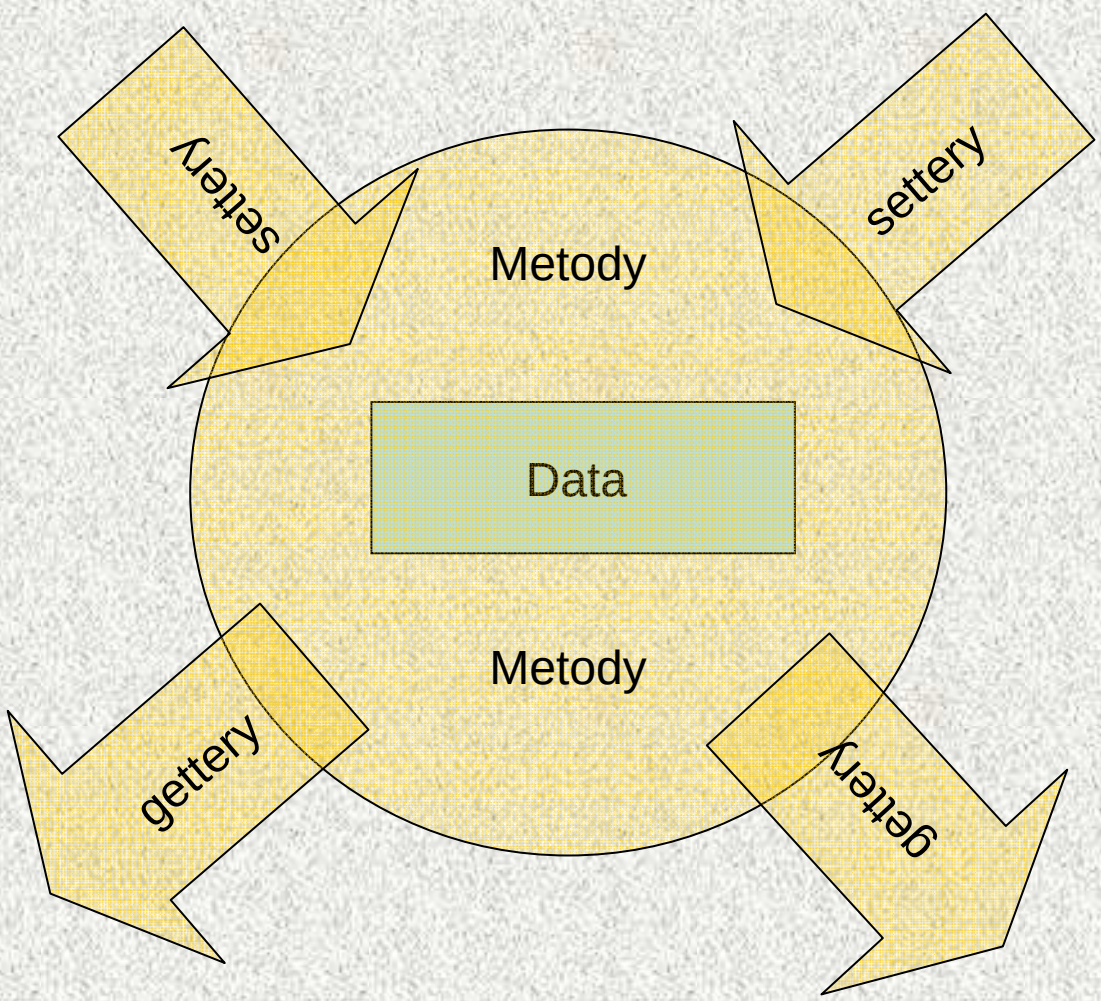
- accessory = settery + gettery
- public metody, které umožňují přístup a nastavení private či protected atributů objektů

```
public class GrO{
   private int poziceX;
...
  public void setPoziceX(int newX){
   if(newX>0)poziceX = newX;
  }
  public int getPoziceX(){
      return poziceX;
  }
  ...
 }
```

setter
- jméno = “set” + jméno atributu
- atribut nesmí být public – Proč?
- kontroluje přípustnost hodnot, aktualizuje závislé hodnoty, např. překreslí objekt na novou pozici

getter
- jméno = “get” + jméno atributu
- vrací hodnotu atributu

# Zapouzdření



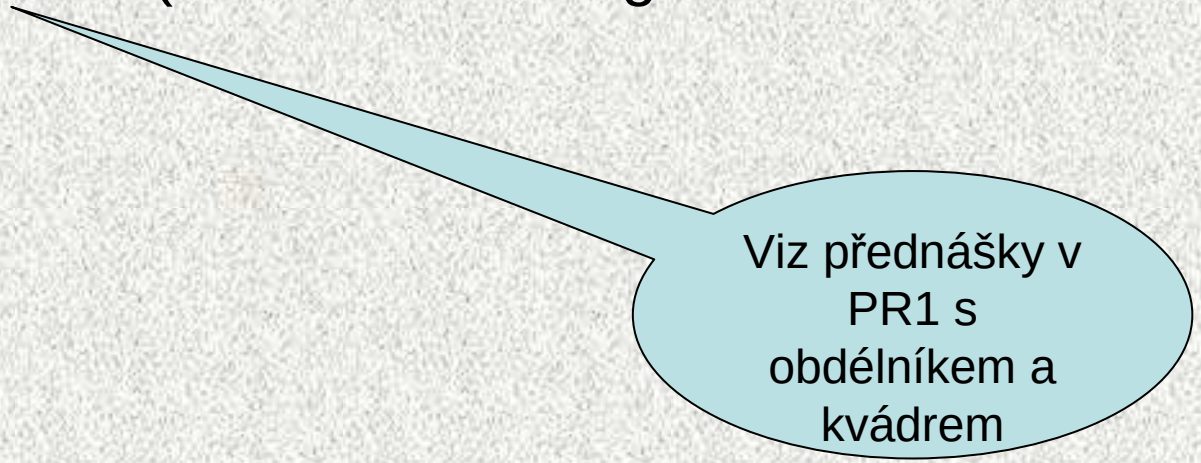
# Dědičnost

Dědičnost:

- ISA vztah (Peter is a student. Auto je dopravní prostředek.)
- vztah nadtřída – podtřída, předek – potomek
- předek shromažďuje společné vlastnosti a předepisuje či i implementuje chování

Kompozice

- HAS vztah (A car has an engine - Auto má motor)



# Polymorfizmus

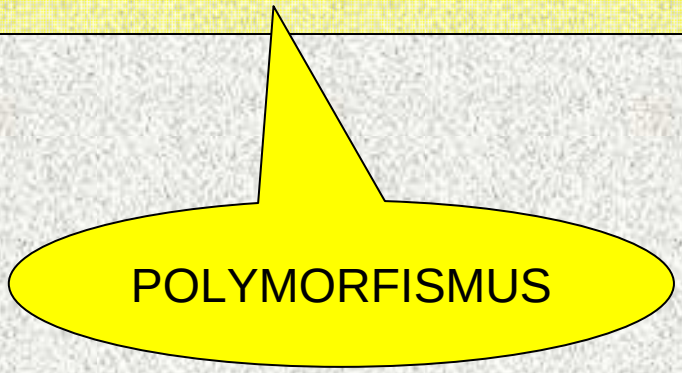
- Polymorfizmus – vícetvarost,
  - základní vlastnost objektového přístupu
  - základní princip polymorfismu:
    - schopnost metody pracovat („přizpůsobit se“) podle typu objektu, na který působí
  - příklady:
    - *„točit volantem“*,
    - *„zaplatit“*
    - *„vypni“*,

# Polymorfismus - příklad

- Co mají zobrazené třídy společné?
- SPÍNAČ!!!
  - lze “spustit” zařízení
  - typ spuštění závisí na typu objektu

# Polymorfismus – abstraktní třída

- Jedno řešení je použití abstraktní třídy (AT)
  - AT sloužila pro specifikaci společných vlastností
    - AT zajistila jednotné pojmenování společných metod
    - při zachování specifického chování dotčeného objektu




často nepoužitelné!
přednost má interface

# Abstraktní třída, abstraktní metody

Abstrakce (abstraktní třídy) - **přínos objektového přístupu**

- Příklad: grafické objekty - abstraktní metody

```
abstract void otoc();
abstract void zmensi(); …
```

- společný předek tříd, jejichž (některé) metody vyžadujme naprogramovat podle potřeb příslušných podtříd
- implementace je **přenechána následníkům resp. implementace je vynucena**
- **abstraktní třída může obsahovat datové složky a neabstraktní metody**
- **třídy mají společnou vlastnost vyjádřenou jako abstraktní metoda**
- **předpoklad systematického polymorfismu**
  - **stejný název metody, různá funkce pro různé objekty**
  - **lze vytvořit referenci na abstraktní třídu,** nikoli její instanci!
- metody jsou označeny **`abstract`,** mají svůj typ, ale nemají parametry ani tělo
- třída s alespoň jedinou abstraktní metodou je abstraktní třídou, označená rovněž **`abstract class`**
- přeprogramovávat je povinné jen abstraktní metody

# Abstraktní třída - příklad

```
abstract class Rodic {
public int i;
  abstract int znasob();
  void setI(int noveI) { i = noveI; }
}
class Potomek1 extends Rodic {
  int znasob() { return i * 2; }
}
class Potomek2 extends Rodic {
  int znasob() { return i * 3; }

public static void main(String[] args) {
Rodic pot;
pot = new Potomek1();
pot.setI(3);
System.out.println("Hodnota je: " + pot.znasob());
pot = new Potomek2();
pot.setI(3);
System.out.println("Hodnota je: " + pot.znasob());
}}
```

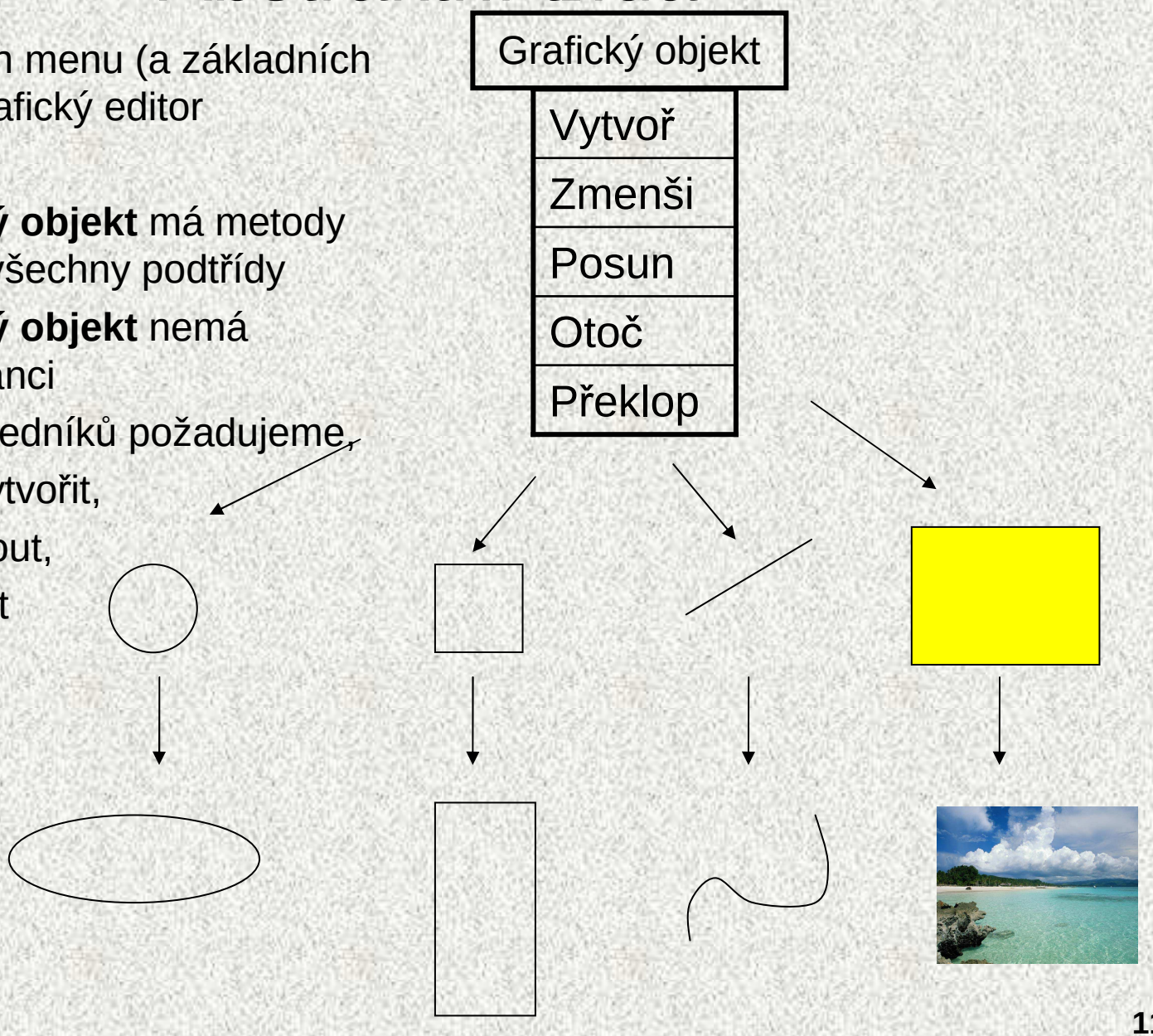
Výstup:

```
Hodnota je: 6
Hodnota je: 9
```

# Abstraktní třída

- Příklad – návrh menu (a základních funkcí ) pro grafický editor

- Třída **Grafický objekt** má metody společné pro všechny podtřídy
- Třída **Grafický objekt** nemá konkrétní instanci
- Od všech následníků požadujeme, aby se uměly vytvořit, zmenšit, posunout, otočit a překlopit



# Abstraktní třídy a polymorfizmus

- Pomocí referenční proměnné předka lze využívat i metody potomka
  - Lze použít referenční proměnnou na abstraktní třídu
  - Často se využívají abstraktní metody k definici „universálního“ předka
- Jasně nadefinujeme, jakou signaturu musejí mít některé metody následníků pro jednotné ovládání, donutíme programátory to respektovat (nebo rozhraním – viz dále)
  - Možnost rozšiřování, není nutný žádný `switch`
  - Použití abstraktní třídy není nutné, kořenová třída nemusí být abstraktní
- Jiná možnost pro polymorfismus - interface

# Rozhraní – `interface`

- Co mají zobrazené třídy společné?
- SPÍNAČ!!!
- Je možné najít společného předka?
- Není, resp. bylo by to nepřirozené!!!
- Řešení – rozhraní, interface!!!

# Rozhraní - `interface`

- Java neumožňuje vícenásobnou dědičnost (problematická záležitost)
  - řešení rozhraní (`interface`)
- Koncepčně můžeme interface považovat za totálně abstraktní třídu, která může obsahovat pouze:
  - abstraktní metody
  - proměnné třídy (implicitně `public static final`) – tedy konstanty
- Interface je konstrukce, která definuje vlastnosti třídy výčtem jejích instančních metod

- Deklarace rozhraní = deklarace hlaviček metod bez implementace, podobně jako v abstraktních metodách
- Třída, která chce rozhraní použít musí všechny metody rozhraní implementovat (překrýt)

# Použití rozhraní, implementace

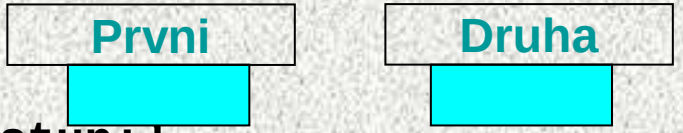
- **Rozhraní jsou implementována třídami**

- Objekty tříd, které implementují stejné rozhraní jsou "zaměnitelné" tímto rozhraním obdobně, jako jsou zaměnitelné hardwarové prvky se stejným rozhraním

- Rozhraní má tyto vlastnosti (na rozdíl od abstraktní třídy):
  - nedeklaruje žádné proměnné
  - třída může implementovat více rozhraní
  - rozhraní nesouvisí s dědičností tříd, s hierarchií tříd
    - různé třídy a implementace téhož rozhraní !
  - nevynucuje „příbuzenské“vztahy
  - vnucuje dovednosti těm, co by toho měly být schopni
    - příklad **`Serializable !! - viz PR1`**

# Rozhraní - příklad

```
interface Inter {
 public void jednotnyVystup();
}
class Prvni implements Inter {
int data;
Prvni(int vstup) { this.data = 14 * vstup;}
public void jednotnyVystup (){
  System.out.println("Data jsou ciselna: " + this.data);}
}
class Druha implements Inter {
String data;
Druha(char vstup) { this.data = "string " + vstup;}
public void jednotnyVystup (){
  System.out.println("Data jsou retezec: " + this.data);}
}
public class Test_rozhrani{
        public static void main(String[] args) {
          new Prvni(14).jednotnyVystup();
          new Druha('g').jednotnyVystup();
        }
      }
```




Data jsou ciselna: 196

Data jsou retezec: string g

# Použití rozhraní jako referenční proměnné

- Proměnnou rozhrani je možné jako referenční proměnnou pro reference na instance tříd, které toto rozhraní implementují
- Je možné volat takto metody rozhraní, nikoli metody tříd!

```
public class Test_rozhrani{
public static void main(String[] args) {
 Inter i;
 Prvni p = new Prvni(14);
 i=p;
 i.jednotnyVystup();
 Druha d = new Druha('g');
 i=d;
 i.jednotnyVystup();        }
 }
```

# Oblasti použití rozhraní

- vnucení metod třídě bez nutnosti zařazení do hierarchie
- vytváření "vícenásobného" dědění
- nalezení podobných "dovedností" pro třídy různých hierarchií
    - mohly vzniknout děděním tříd z knihovny, jiných autorů ...
    - společný předek by byl „vykonstruovaný“

- Trend - použití rozhraní namísto abstraktních tříd
    - dodávka: neabstraktní třída + rozhraní (popisuje všechny metody)

Pozn: Abstraktní třída může „implementovat“ rozhraní bez skutečné implementace těla metod rozhraní

# Polymorfizmus a rozhraní - příklad I

- Je-li nutné přistupovat k třídám různých hierarchií stejným způsobem a nelze-li vytvořit společného předka, pak použijeme rozhraní

```
interface Vazitelny {
  public void vypisHmotnost();
}
class Clovek implements Vazitelny {
  int vaha;
  String profese;
  Clovek(String povolani, int tiha) {
    profese = new String(povolani);
    vaha = tiha;}
  public void vypisHmotnost() {
    System.out.println(profese + ": " + vaha);}
  public int getHmotnost() { return vaha; }
}
```

# Polymorfizmus a rozhraní - příklad II

```
class Kufr implements Vazitelny {
  int vaha;
  Kufr(int tiha) { vaha = tiha; }
  public void vypisHmotnost() {
    System.out.println("kufr: " + vaha);}
}
```

# Polymorfizmus a rozhraní - příklad III

```
public class PolymRozhra {
public static void main(String[] args) {
int vahaLidi = 0;
Vazitelny[] kusJakoKus = new Vazitelny[3];
kusJakoKus[0] = new Clovek("programator", 100);
kusJakoKus[1] = new Kufr(20);
kusJakoKus[2] = new Clovek("modelka", 51);
for (int i = 0;  i < kusJakoKus.length;  i++) {
kusJakoKus[i].vypisHmotnost();
if (kusJakoKus[i] instanceof Clovek == true)
vahaLidi += ((Clovek) kusJakoKus[i]).getHmotnost();
}                  //nutno přetypovat!!
System.out.println("Ziva vaha: " + vahaLidi);
  }}
```

programator: 100
kufr: 20
modelka: 51
Ziva vaha: 151

# Srovnání rozhraní a abstraktní třídy

| interface | abstraktní třída |
|---|---|
| obsahuje pouze abstraktní metody | může obsahovat i neabstraktní metody |
| obsahuje pouze konstanty | může obsahovat libovolné položky |
| nemusí mít předchůdce | vždy má nadtřídu (např. `object`) |
| třída může implementovat více rozhraní | třída může rozšiřovat právě jednu nadtřídu |

# Třída x abstraktní třída x rozhraní

Rozhraní = interface

„totálně abstraktní třída“

- nemůže obsahovat atributy
- může obsahovat pouze konstanty
- nemůže mít implementované metody
- všechny jeho metody jsou abstraktní metody
- nemůže mít konstruktor
- nelze vytvořit instanci

Třída = class

- nesmí obsahovat abstraktní metody

Abstraktní třída = abstract class

„nedodělaná třída“

- může obsahovat atributy
- může mít implementované metody
- může mít abstraktní metody
- může mít konstruktor
- nelze vytvořit instanci (konstruktor lze volat z potomka)

GrO – třída, nebo abstraktní třída, nebo rozhraní?

# Výčtové typy (Java 5 )

- Výčtové typy jsou speciální třídy zavedené pro větší bezpečí a pohodlí, v nejjednodušší variantě se definují příkladně takto:

```
enum Day {SUN, MON, TUE, WED, THU, FRI, SAT;}
```

- a mají ordinální čísla 0 .. 6.

Vypíší se takto:

```
for ( Day d : Day.values( ) )
System.out.println( d.ordinal( )+ "   " +d.name( )  );
```

Užijí se např. takto:

```
Day d =  Day.MON;
switch ( d ) {
 case SAT:
 case SUN:
 case MON:  ...    // No work
 case FRI:  ...    // Partial work
 default:     ...    // Work
}
```

```
for ( <Typ> <parametr> :<kontejner>
```

**Iterátor, viz kolekce 7. přednáška**

# Příklad s atributy

```
public enum Day {
SUN ( "Sunday",      "dimanche",    new Point(0,0) ),
MON ( "Monday",      "lundi",       new Point(1,0) ),
TUE ( "Tuesday",     "mardi",       new Point(2,0) ),
WED ( "Wednesday",   "mercoledi",   new Point(3,0) ),
THU ( "Thursday",    "jeudi",       new Point(4,0) ),
FRI ( "Friday",      "vendredi",    new Point(5,0) ),
SAT ( "Saturday",    "samedi",      new Point(6,0) ),
    ;
public final String en;
public final String fr;
public Point  where;

private Day( String en, String fr, Point where ) {
this.en=en;  this.fr=fr;  this.where = where;
    }
}
```

Konstruktor je private – proč?

Důležité – příklad s atributy

# Příklad s atributem a metodami

```
public enum Month {
JAN (31),
FEB (-1) {public int days( int yy ) { return  yy%4==0 ? 29 : 28;}},
MAR (31), APR (30), MAY (31), JUN (30),
JUL (31), AUG (31), SEP (30), OCT (31), NOV (30), DEC (31) ;

private final int days;                              // hidden
private  Month(int days) { this.days = days; }    // konstructor
public int days(int year) { return days; }
public static int totalDays(int year){return year%4==0  ? 366 : 365;}
 }
-------
for (Month m  : Month.values())
System.out.println(m.ordinal() + " " + m.name() + " "+m.days(2000));
System.out.println("  " + Month.totalDays(1000));
```

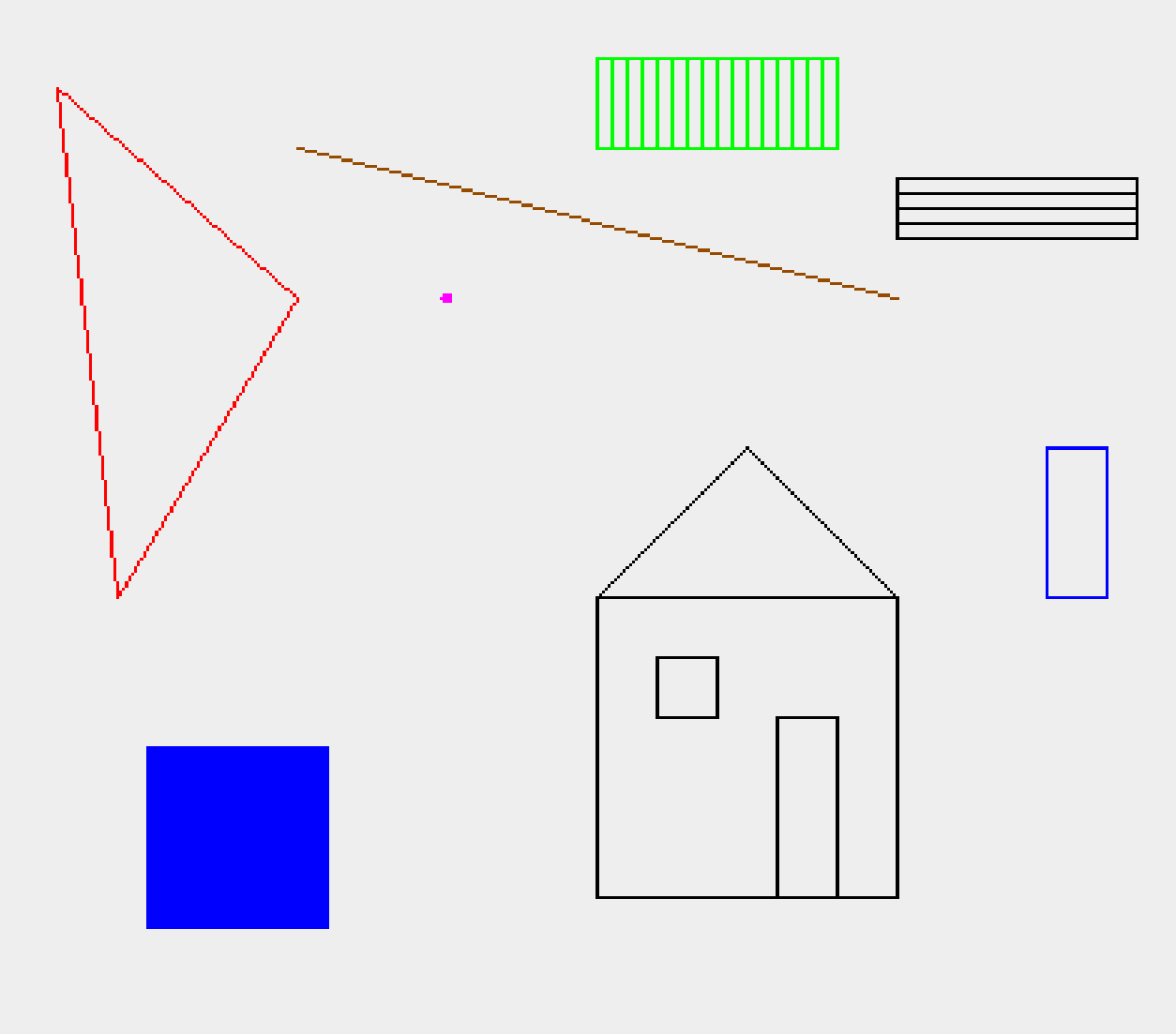
Nedokonalý výpočet, pouze ilustrace
únor má zastíněnou metodu days,
speciální chování

# Polymorfizmus, zadání příkladu

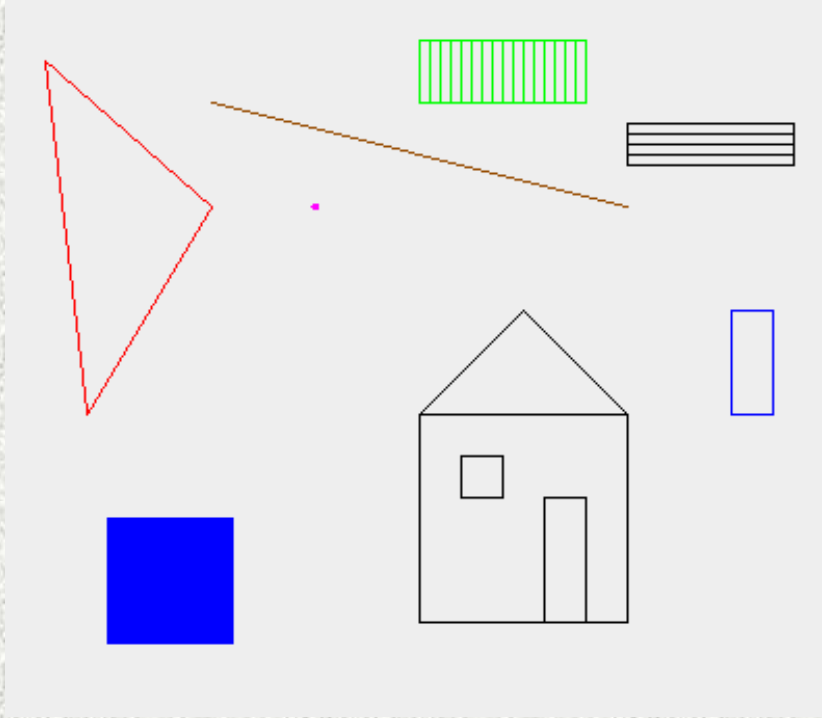
- Vytvořme základ vektorového kreslícího programu
- Vektorový editor ?
- Jaký jiný?
  - Rastrový
- GUI – bude příští přednášku
- Vytvoříme hierarchii objektů, které bude možné kreslit a budeme se věnovat jejich vztahům

GUI - Graphical user interface,
grafické uživatelské rozhraní

# Základní objekty



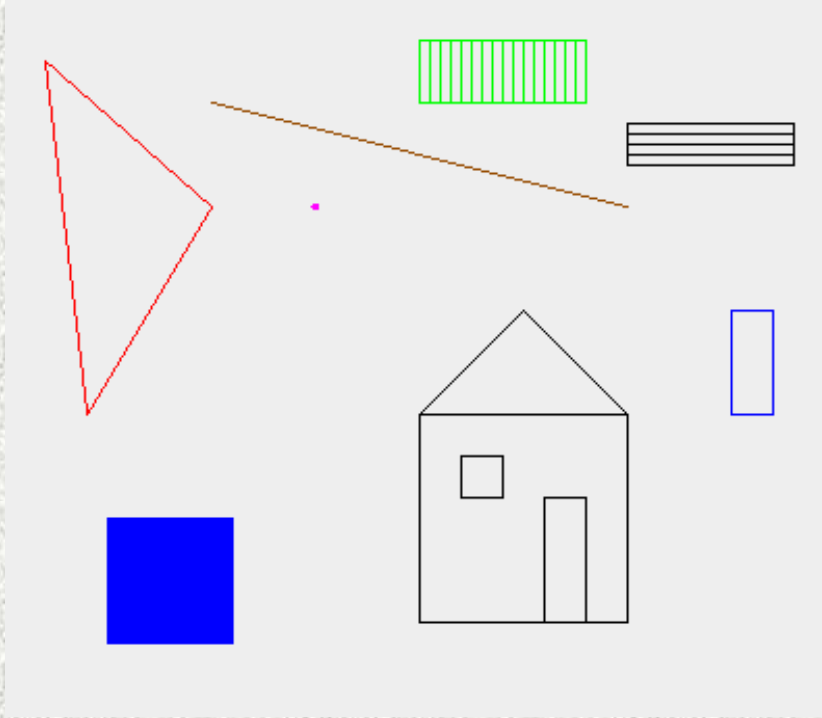
# Objekty



- obdélník
- bod
- úsečka
- trojúhelník
- domeček
  - komplexní, složený objekt
- mnohé další .....

Objekty v editoru budou odpovídat instancím tříd, které popíší vlastnosti a chování těchto objektů.

# Společné vlastnosti



Vlastnosti = atributy objektů

- umístění v obrázku
  - pozice středu, těžiště
  - pozice levého horního bodu
  - pozice charakteristického bodu
- barva
- velikost?
  - co je to velikost pro bod?
  - pro obdélník?
  - pro trojúhelník?
  - ➢ specifické pro daný typ objektu
- typ výplně
  - jakou má výplň bod?
  - jakou úsečka?
  - ➢ Jde o vlastnost společnou jen některým objektům

# Popis chování = metody

- metody pro nastavení a získání atributů objektů
  - settery: nastavení barvy, pozice, ...
  - gettery: zjištění šířky, výšky, ...
- metody pro komplexní práci s objekty
  - posun
  - rotace
  - zjištění vzdálenosti od vybraného bodu

# Společná nadtřída všech grafických obj.

- název např. GrO
- společné vlastnosti všech potomků :
  - barva
  - pozice
  - a jistě i další (úhel pootočení, měřítko zvětšení, ...)
- společné chování všech potomků:
  - umí nastavit a získat barvu
  - umí se posunout
  - umí se vykreslit

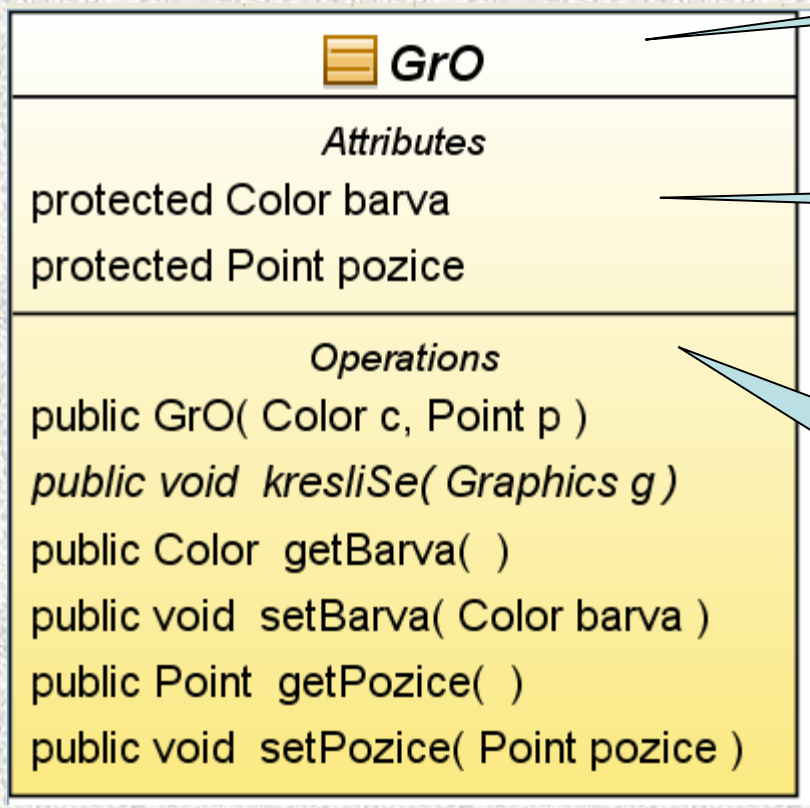
# UML, Unified Modeling Language

- grafický jazyk pro vizualizaci, specifikaci, navrhování a dokumentaci programových systémů
  - strukturní diagramy:
    - diagram tříd
    - diagram komponent
    - composite structure diagram
    - diagram nasazení
    - diagram balíčků
    - diagram objektů, též diagram instancí
  - diagramy chování:
    - diagram aktivit
    - diagram užití
    - stavový diagram
    - diagramy interakce: sekvenční diagram,diagram komunikace,interaction overview diagram,diagram časování

Pro informaci

# GrO – obecný grafický objekt

diagram třídy:




Název třídy, kurzíva = abstraktní třída

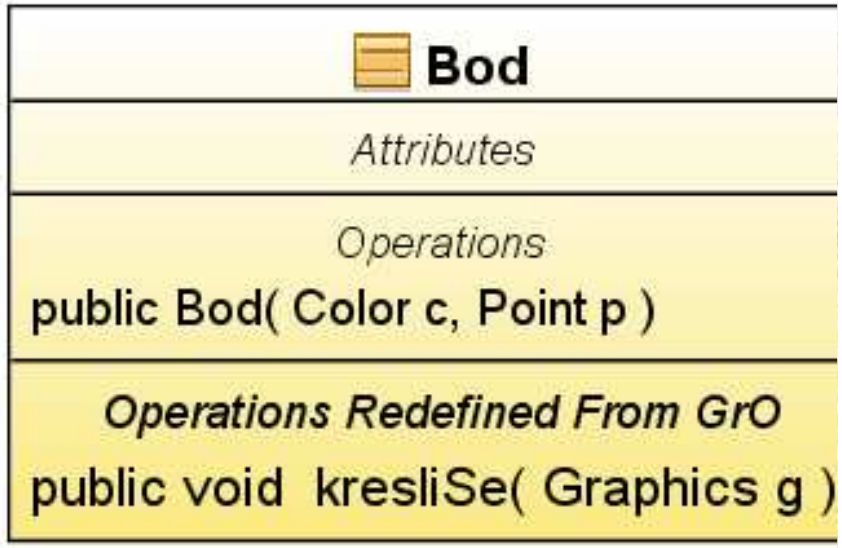
Atributy

Metody
Název metody kurzívou = abstraktní metoda

# Bod

```
public class Bod extends GrO{
public Bod(Color c, Point p){
super(c,p);
}


@Override
public void kresliSe(Graphics g) {
Color c = g.getColor(); //uloz si barvu pera
g.setColor(barva); //nastav svou barvu
g.fillOval(pozice.x-2, pozice.y-2, 4,4);
        //bod je takove malé kolecko
g.setColor(c); //obnov barvu
    }
}
```



# Předek či potomek

- Je Usecka potomek Bodu?
  + Usecka rozšiřuje Bod o jeden jeho konec
  - Usecka nevyužije ani jednu metodu Bodu
  - ISA vztah: ?Usecka je Bod? – **NE** → Usecka není potomkem Bodu

- Je Obdelnik potomkem Usecky?
  – ISA vztah → **NE**

- Je Obdelnik potomkem Ctverce nebo naopak?
  – Obdelnik rozšiřuje Ctverec o další rozměr, ale není Ctverec
  – Ctverec je Obdelnik, který má sířku i výšku stejnou

# Doporučení pro návrh tříd

Třída

- jasně definujte povinnosti/zodpovědnost
  - měla by dělat jednu věc a dělat ji dobře
    - Třída *SeznamGrO* představuje seznam grafických objektů, reprezentovaných třídou GrO a jejími potomky.
    - pozor na „božské“ třídy, které všechno ví a všechno umí
  - uvažujte na úrovni abstraktních datových typů – ADT

    ADT: zakrývá implementační detaily, související věci dává k sobě, samopopisné rozhraní umožňuje eliminovat a lépe odhalovat chyby
- Specifikujte kontrakt objektu
  - invarianty (vlastnosti neměnné během vykonávání algoritmu)
  - interakce s okolím
  - kontrakty jednotlivých metod

# Dědičnost - výhody

- Jednotné zacházení s více třídami
  - všechny GrO mají barvu a metodu pro její změnu – stejné chování
  - kolekce GrO bude obsahovat různé objekty, ale všechny se umí vykreslit – různé implementace, stejná funkce
  - mechanizmus pro realizaci polymorfizmu

- Omezení duplicit kódu
  - společné rysy skupiny tříd sdílejí definici a popř. implementaci (např. barva a pozice u GrO
  - základní třída definuje společné rysy na jednom místě
  - odvozené třídy definují své specifické rysy

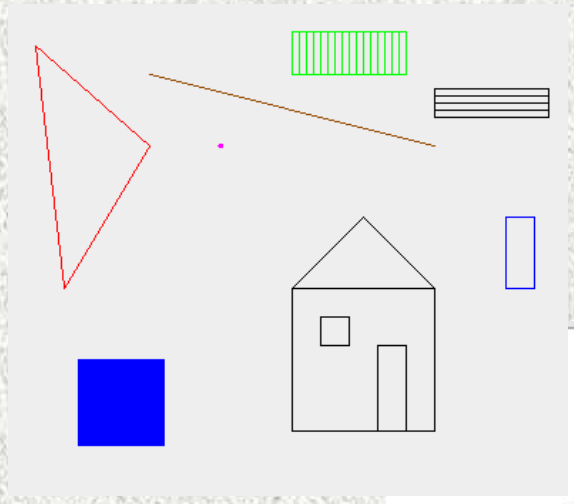
# Použití dědičnosti

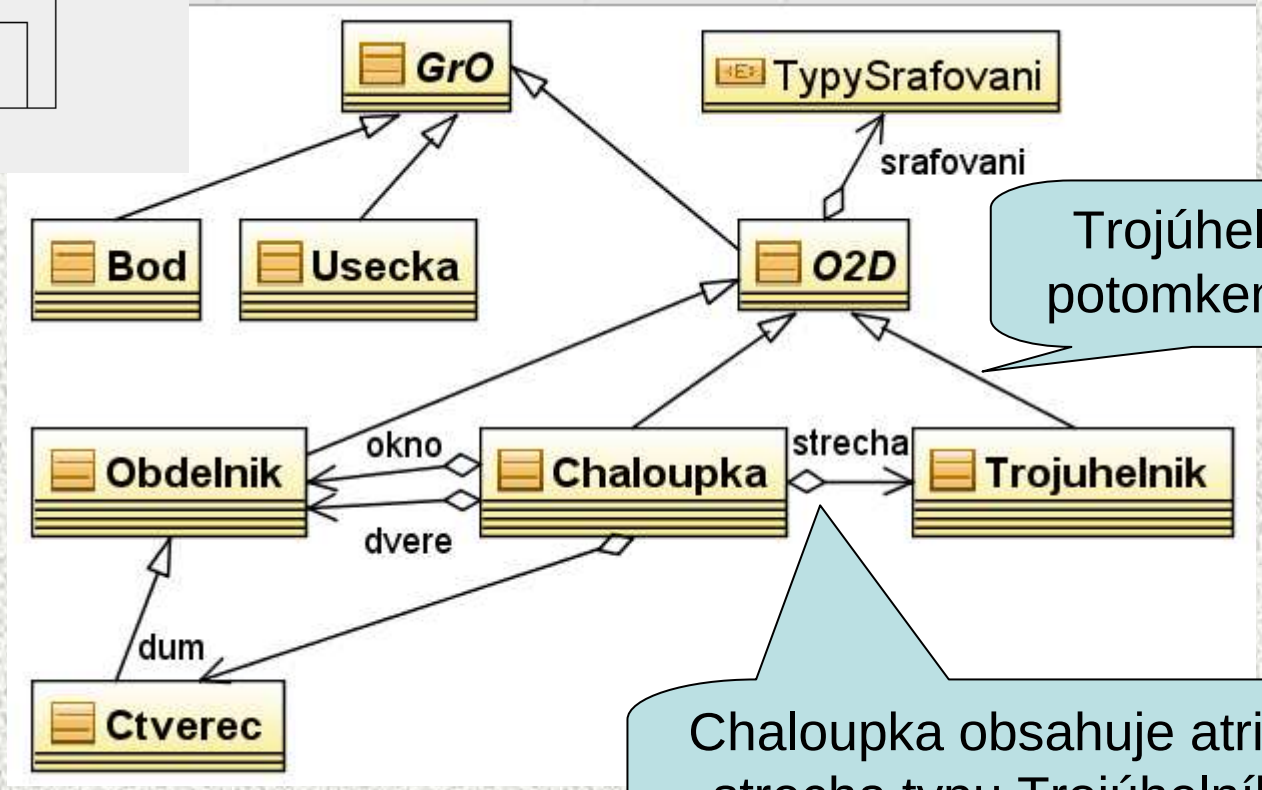
- Substituční princip (Barbara Liskov)
  - použití dědičnosti pouze pokud je (ISA) podtřída skutečně specializací nadtřídy
  - všude, kde lze použít nějaká třída, musí jít použít i její podtřída, aniž by uživatel poznal rozdíl
  - netýká se jen syntaxe, ale i sémantiky (tu je nutné vhodně popsat)
  - podtřída musí dodržovat kontrakt nadtřídy
- Vztah (ISA) musí být trvalý

# Zásady návrhu dědičnosti

- Zvažte:
  - Jaká má být viditelnost atributů, metod a dalších prvků?
  - Které metody mají být virtuální (lze je přepsat v potomkovi, not final)?
  - Které metody mají být abstraktní? Má metoda kresliSe v GrO vypsat chybu nebo být abstraktní a kreslení delegovat na potomka?
  - Nesnížíme příliš flexibilitu pokud zakážeme dědičnost (final class)?
- Hierarchie tříd
  - společné věci přesunout co nejvýše
  - každá abstraktní třída a rozhraní by měly mít alespoň dva potomky
  - hluboká hierarchie zvyšuje paměťové a časové nároky, snižuje přehlednost – doporučená hloubka asi 3 úrovně.

# Diagram tříd

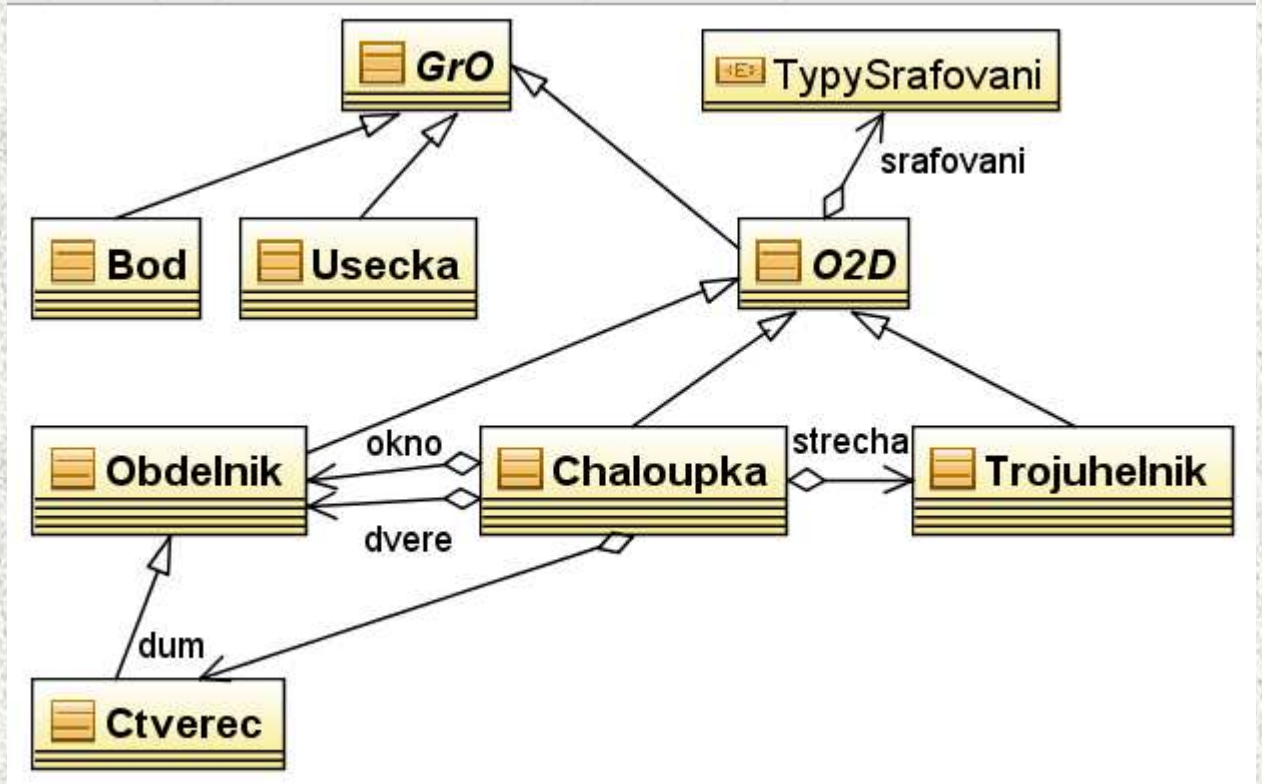





Trojúhelník je potomkem O2D

Chaloupka obsahuje atribut strecha typu Trojúhelník - kompozice

# Diagram tříd




Kam bychom zařadili

- textový řetězec?
- mnohoúhelník?
- městečko?

# Ukázka vytváření diagramů tříd v Netbeans

- Instalace pluginu
- Reverzní inženýrství
- další možnosti ...

# Appendix

# Abstraktní třída java.lang.Enum

```
public abstract class Enum <E extends Enum<E> >
                    implements Comparable<E>, Serializable
```

- **Comparable** - pro porovnání výčtových konstant dle ordinálních čísel.

Definuje tyto public metody:

```
static  T [ ] values()  - vrátí pole
static <T extends Enum<T>> valueOf(String name)
static <T extends Enum<T>> valueOf(Class enum T, String name)

String name( ) – vrátí jméno konstanty
int ordinal( ) – vrátí pořadové číslo tj. index položky
Class <T> declaringClass( )
int compareTo( T  o )
```

Pro informaci

# Syntax výčtových typů

- Výčtové typy jsou potomky abstraktní třídy java.lang.Enum (ta je přímým potomkem java.lang.Object ), z třídy v Enum však nelze přímo dědit.

```
[public] enum Name [implements Interface1, Interface2, ... ]  {
XX[(a, b,...)][{ nestatické členy anonymní třídy $1}],
                                  // statický kontext
YY[(a, b,...)][{ nestatické členy anonymní třídy $2 } ],
     ...  ;                  // Výčtové  konstanty musejí být první
                             //  - další statický kontext kdekoli dále.
[[private] Name (Type1 a,Type2 b, ... )  {
 this.a = a; this.b = b; ... } ]
                            // Konstruktorů může být libovolný počet.
... Type1 a;                 // Nestatické atributy všech výčtových
    ... Type2 b = ... ;      // konstant ( nemusejí být finální ani
    ... Type3 q = ... ;                          // skryté ).
    ...
[ [ abstract ] ... m1 ( ... )  ... ]//Nestatické  metody zdědí všechny
[ [ abstract ] ... m2 ( ... )  ... ]//konstanty, avšak lze je přepsat
    ...                              //  v anonymních třídách.
}
```

pro každou konstantu

Pro informaci

- meta symboly