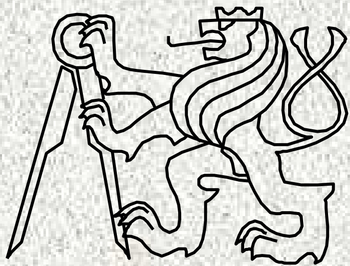
Třídy, polymorfismus, emum



A0B36PR2-Programování 2

Fakulta elektrotechnická
České vysoké učení technické

Obsah přednášky

- Zapouzdření, opakování
- Dědičnost, opakování
- Polymorfismus
  - Abstraktní třída
  - Interface
- Výčtové typy
- Modelování (pro zájemce)

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)

Viz přednášky v PR1 s ředitelem a datem narození

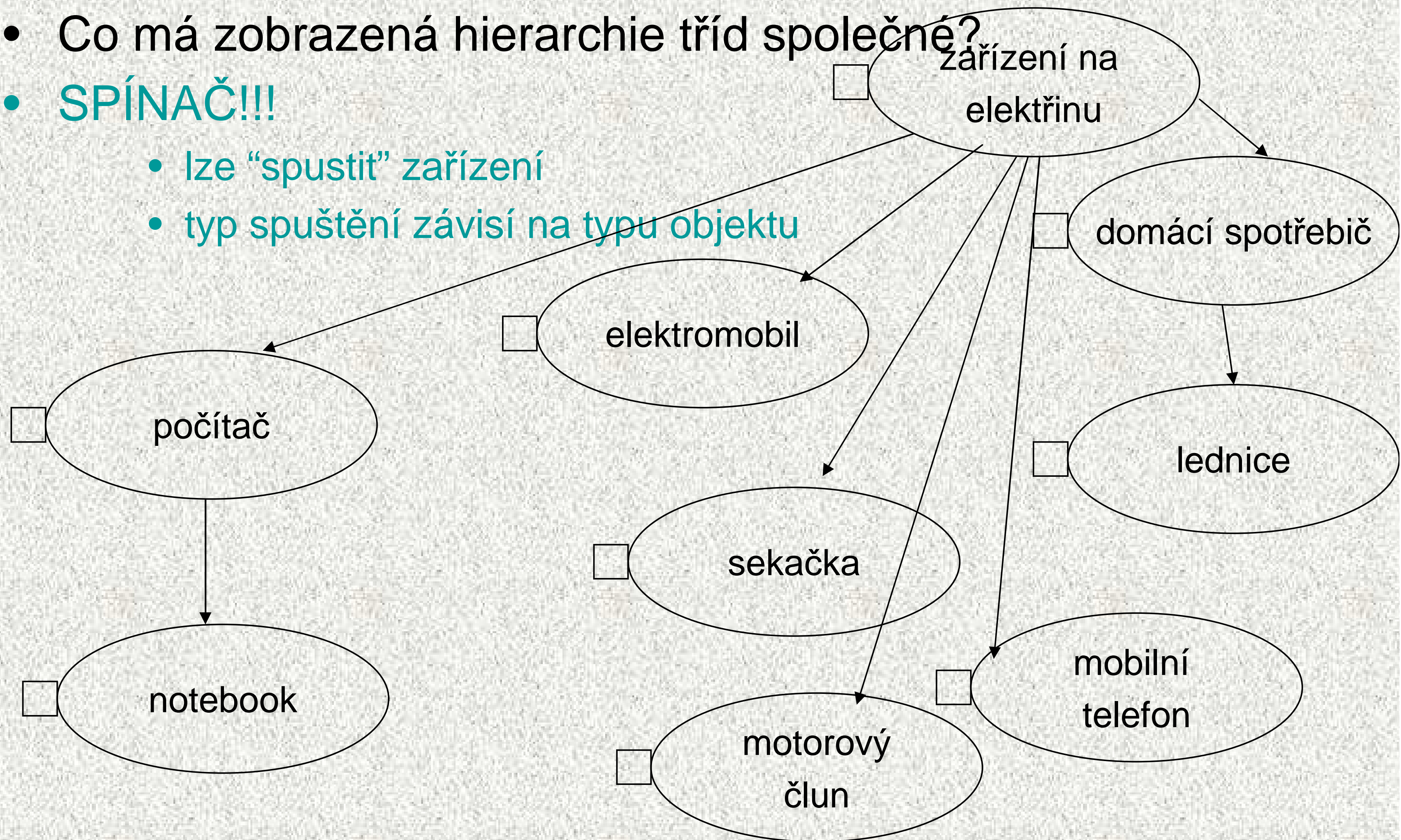
Viz přednášky v PR1 s obdélníkem a kvádrem

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 má zobrazená hierarchie tříd 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 – nutná hierachie tříd
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:

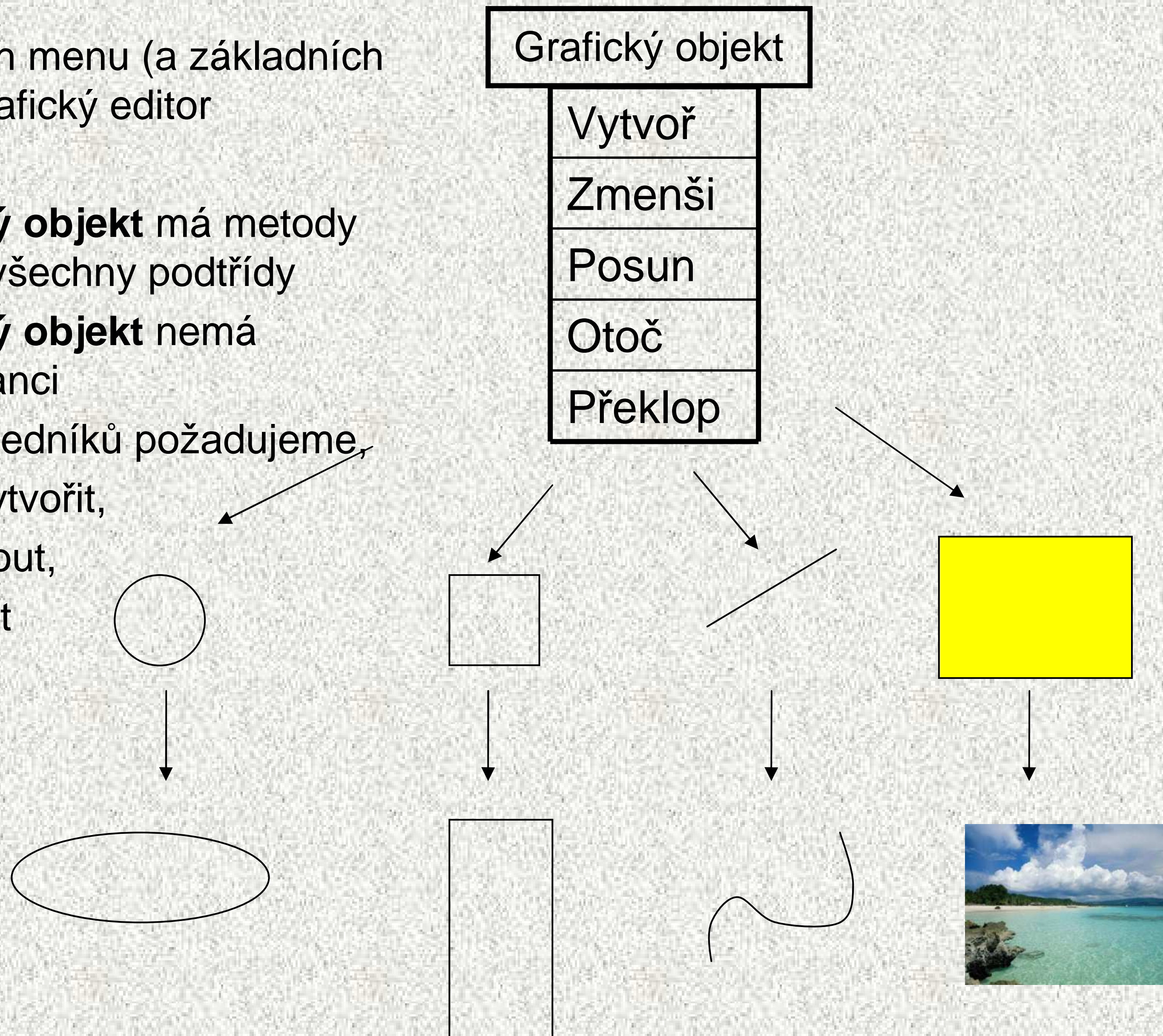
```
Hodnota je: 6
Hodnota je: 9
```

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

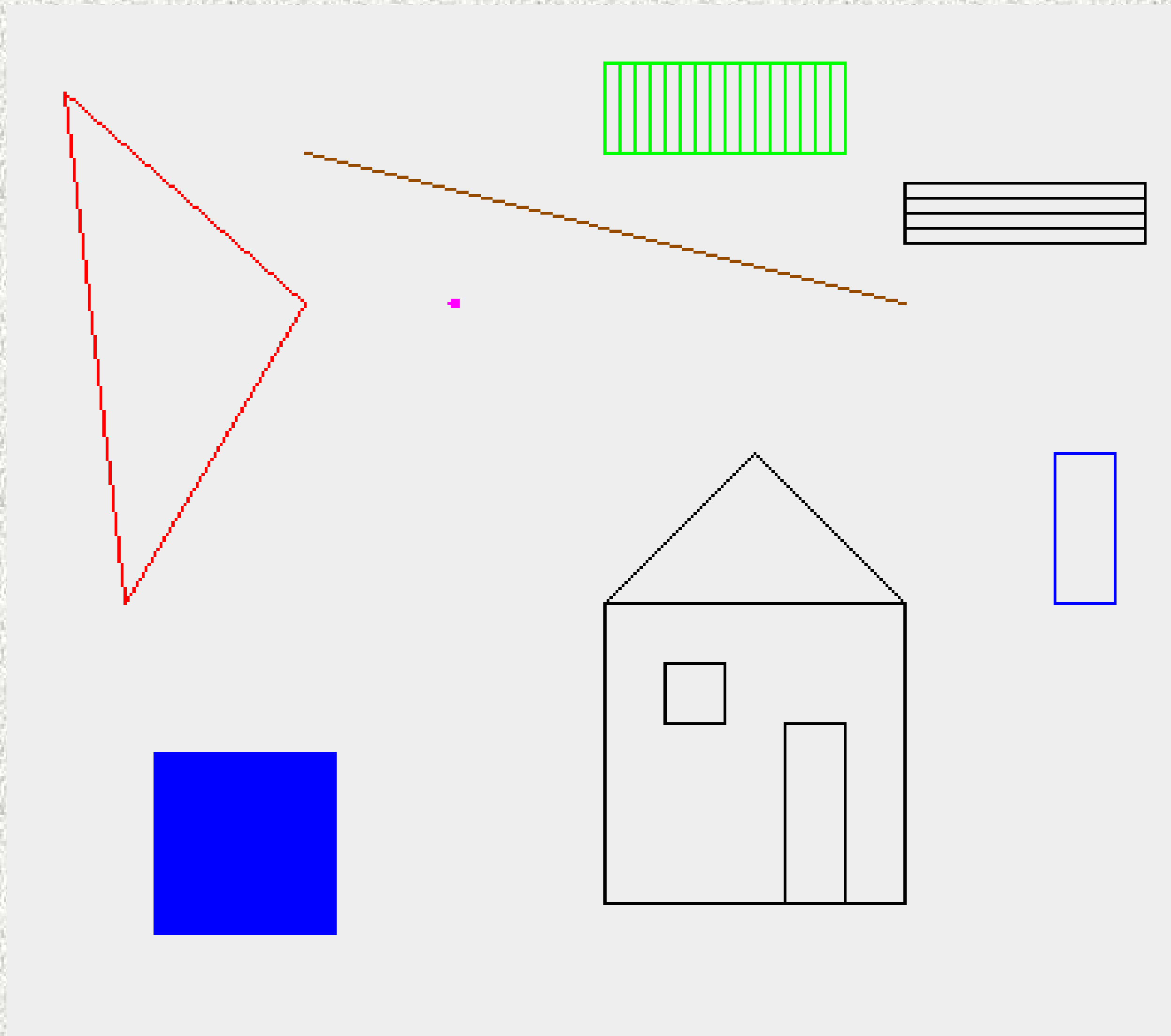
- Pomocí referenční proměnné předka lze využívat i metody potomka
  - Lze použít referenční proměnnou na abstraktní třídu
  - Často se využívají abstraktní metody k definici „universálního“ předka
- Jasně nadefinujeme, jakou signaturu musejí mít některé metody následníků pro jednotné ovládání, donutíme 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

Abstraktní třídy a polymorfizmus - příklad

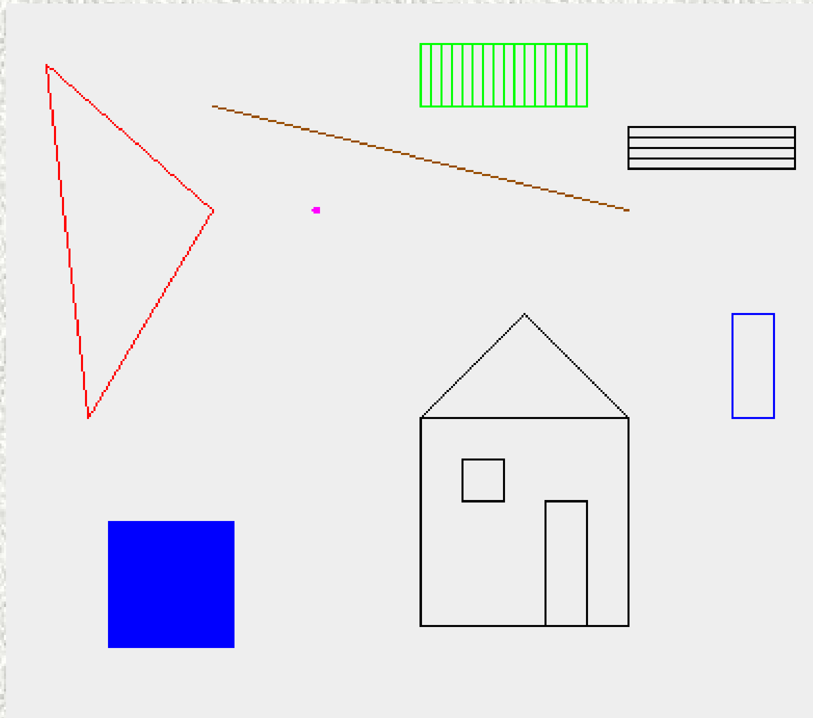
- Vytvořme základ vektorového kreslícího programu
- Vektorový editor ?
- Jaký jiný?
  - Rastrový
- GUI – bude příští přednášku
- Vytvoříme hierarchii objektů, které bude možné kreslit a budeme se věnovat jejich vztahům

GUI - Graphical user interface,
grafické uživatelské rozhraní

Základní objekty



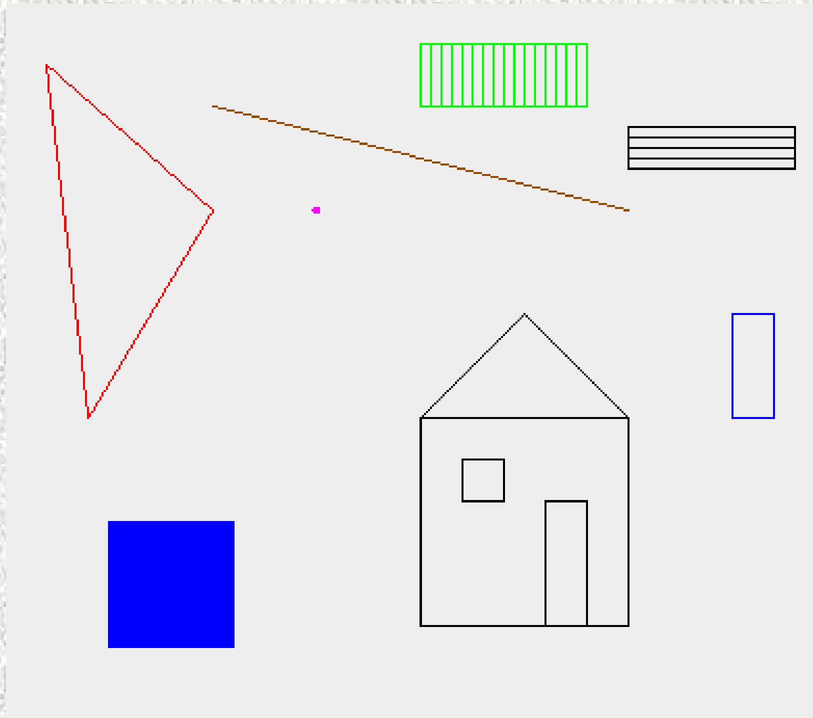
Objekty



- 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



- 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

Vlastnosti = atributy objektů

Popis chování = metody

- metody pro nastavení a získání atributů objektů
  - settery: nastavení barvy, pozice, ...
  - gettery: zjištění šířky, výšky, ...

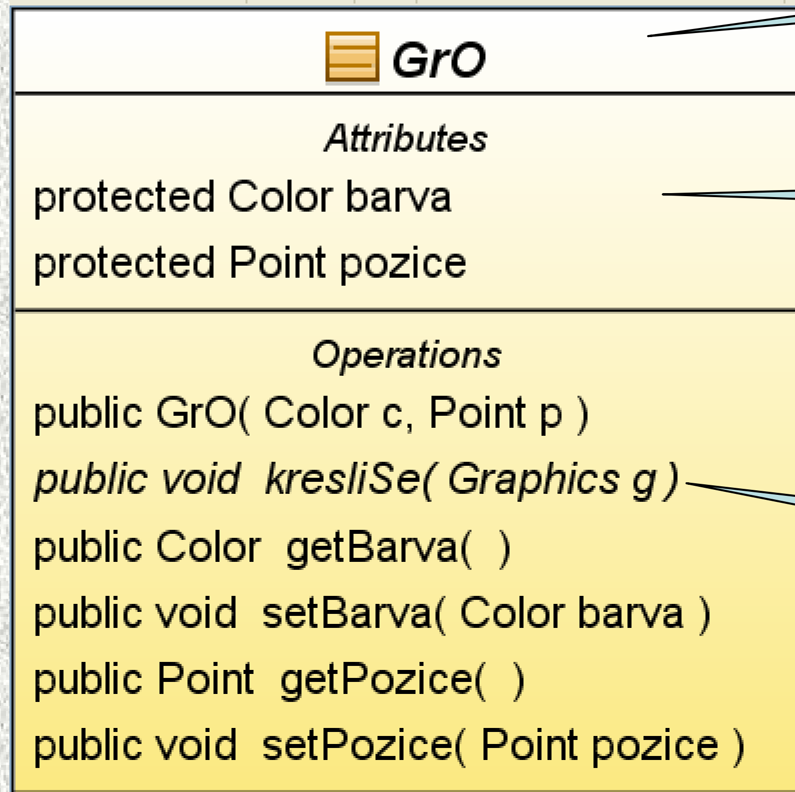
- metody pro komplexní práci s objekty
  - posun
  - rotace
  - zjištění vzdálenosti od vybraného bodu

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***

GrO – obecný grafický objekt

diagram třídy:

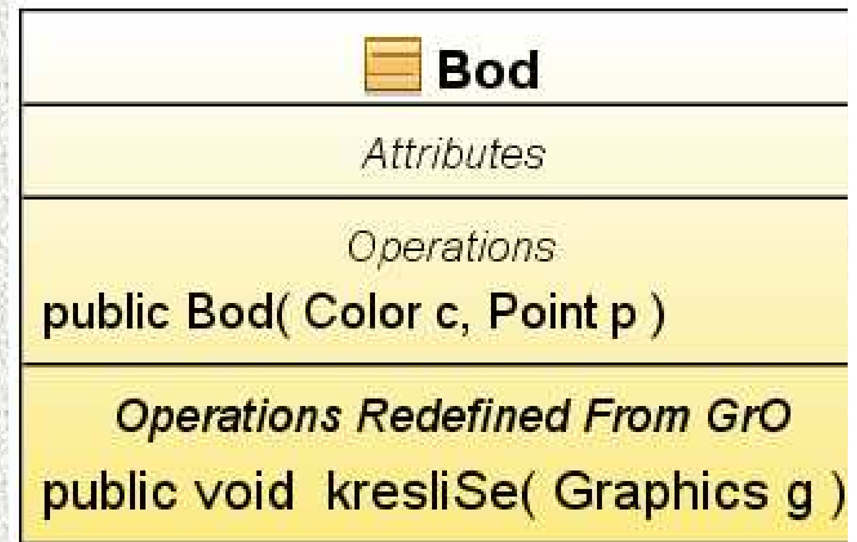


Název třídy, kurzíva = abstraktní třída

Atributy

Metody
Název metody kurzívou = abstraktní metoda

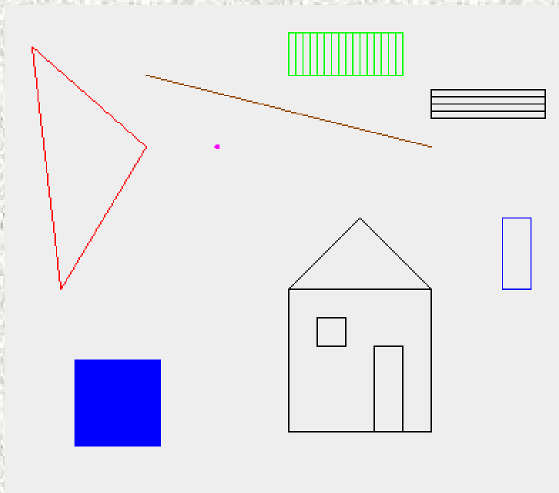
Bod

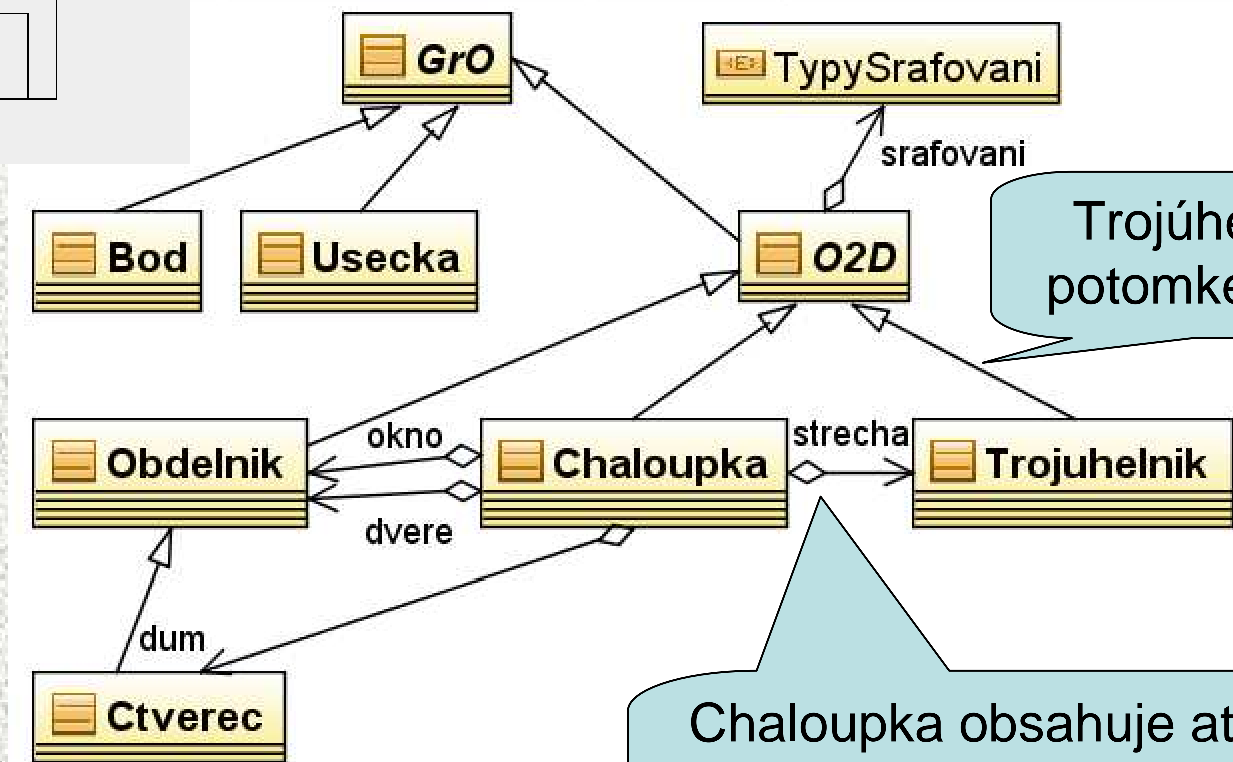



```
public class Bod extends GrO{
public Bod(Color c, Point p){
super(c,p);
}


@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
    }
}
```

Diagram tříd

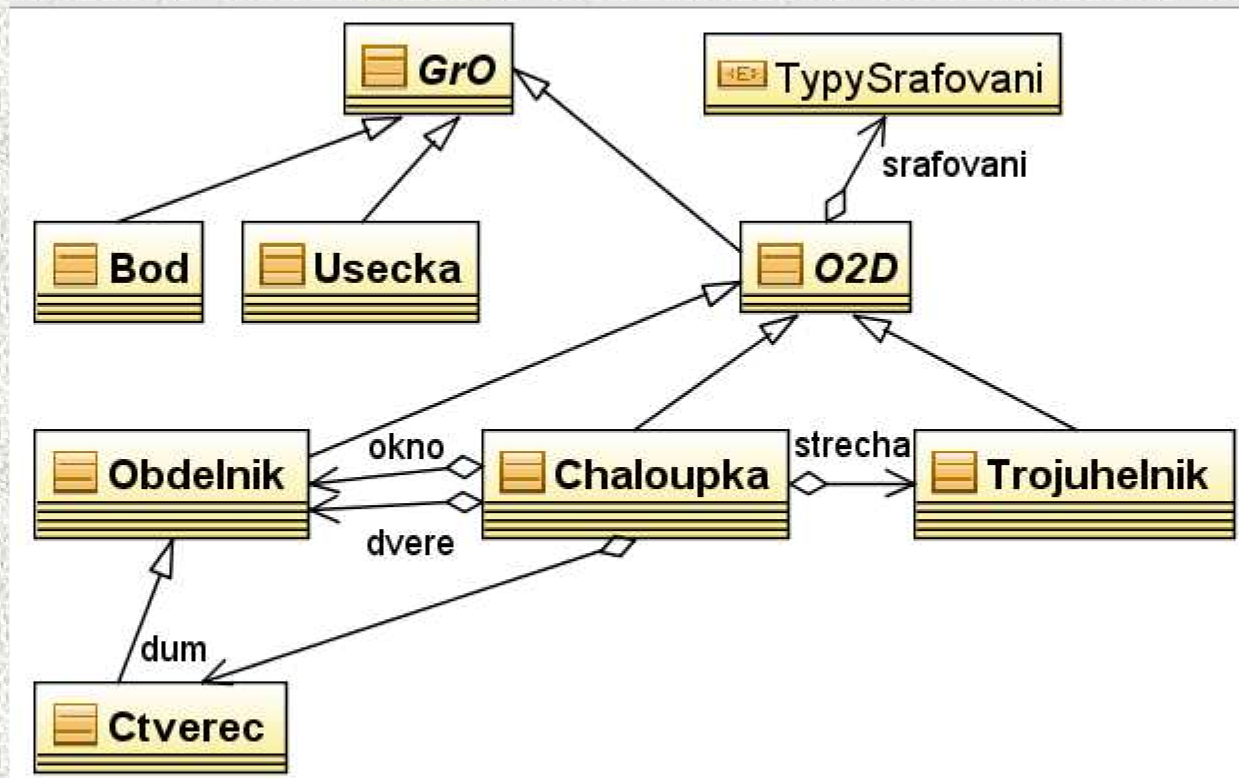




Trojúhelník je potomkem O2D

Chaloupka obsahuje atribut strecha typu Trojúhelník - kompozice

Diagram tříd



Kam bychom zařadili
- textový řetězec?
- mnohoúhelník?
- městečko?

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
  - konstanty (implicitně `public static final`)
- 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 interface, 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 Testrozhrani{
         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 = 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
Živa 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

Ukázka vytváření UML diagramů tříd v Netbeans

- Reverzní inženýrství
  - Pravým tlačítkem na projekt
  - Vybrat Reverse engineering, O.K.
  - rozkliknout v UML projektu "… - Model" záložku „Model"
  - pravým tlačítkem kliknout na balíček z java projektu a zvolit "Create Diagram".
  - Potom vybrat třeba "Class diagram" a mělo by se otevřít okénko

Výčtové typy (Java 5)

- Pojmenované konstanty
- Konečný „malý“ počet konstant, nezáleží na jejich reprezentaci
- Pro jeden účel, konstanty spolu souvisí
- Možnost konstanty „parametrizovat“
- Typové zabezpečení
- Syntaxe:
  - Seznam jednotlivých hodnot daného typu je je možno definovat jejich prostým výčtem, např.:

```
enum Den {PO, UT, ST, CT, PA, SO, NE; }
```

  - Výčet proměnných „konstant“ obsahujících i parametry – viz dále
    - předávané konstruktoru za název deklarované proměnné seznam parametrů
    - třeba definovat konstruktor

Výčtové typy

- Výčtové typy jsou speciální třídy zavedené pro větší bezpečí a pohodlí, v nejjednodušší variantě se definují (příklad):

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

- mají ordinální čísla 0 .. 6.

Vypíší se takto:

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

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

Iterátor, viz kolekce 7. přednáška

```
0   SUN
1   MON
2   TUE
3   WED
4   THU
5   FRI
6   SAT
```

Výčtové typy

```
Den d = Den.MON;

    switch (d) {
        case SAT:
            System.out.println(" sat");
            break;
        case SUN:
            System.out.println(" sun");
            break;
        case MON:
            System.out.println(" No work");
            break;   // No work
        case FRI:
            System.out.println(" Partial work ");
            break;   // Partial work
        default:
            System.out.println(" work");
            break;   // Work
    }
}
```

No work

Výčtové typy 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) ),
     ;
String en;
String fr;
Point  where;

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

Konstruktor je private

Výčtové typy s atributy

```
System.out.println("Streda ve zkratce " + Day.WED + " anglicky
   je " + Day.WED.en + " a francouzky je " + Day.WED.fr);
for (Day d : Day.values()) {
System.out.println(d.ordinal() + "   "
                                + d.name() + " angl."
                                        + d.en + " \tfr."
                                                  + d.fr );
}
```

```
Streda ve zkratce WED anglicky je Wednesday a francouzky je mercoledi
0   SUN  angl.Sunday        fr.dimanche
1   MON angl.Monday         fr.lundi
2   TUE  angl.Tuesday       fr.mardi
3   WED angl.Wednesday   fr.mercoledi
4   THU  angl.Thursday      fr.jeudi
5   FRI    angl.Friday         fr.vendredi
6   SAT   angl.Saturday      fr.samedi
```

Výčtové typy s atributy a metodami

```
public enum Month {
    JAN(31,5),
    FEB(-1,3) {
    @Override
    public int days(int yy) {
     return yy % 4 == 0 ? 29 : 28;
    } },
    MAR(31,3), APR(30,3), MAY(31,3), JUN(30,5),
    JUL(31,3), AUG(31,1), SEP(30,5), OCT(31,4), NOV(30,5), DEC(31,3);

    private final int days; // hidden
    private final int ds; // hidden

    private Month(int days, int ds) {
        this.days = days;
        this.ds = ds;
    }    // konstructor

    public int days(int year) {
        return days;}

    public int ds(int year) {
        return ds;
    }

    public static int totalDays(int year) {
        return year % 4 == 0 ? 366 : 365;
    }
}
```

FEB má zastíněnou metodu days, speciální chování

Výčtové typy s atributy a metodami

```
 public static void main(String[] args) {
 for (Month m : Month.values())
System.out.println(m.ordinal() + "    "
+ m.name() + "    "
+ m.days(1999) + " "
+ m.ds(1222));
}
System.out.println("   " + Month.totalDays(1000));
    }
}
```

```
0   JAN   31 5
1   FEB   28 3
2   MAR   31 3
3   APR   30 3
4   MAY   31 3
5   JUN   30 5
6   JUL   31 3
7   AUG   31 1
8   SEP   30 5
9   OCT   31 4
10   NOV   30 5
11   DEC   31 3
  366
```

Konec

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

Abstraktní třída = abstract class

„nedodělaná třída“

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

Třída = class

- nesmí obsahovat abstraktní metody

Doporučení pro návrh tříd

Třída

- jasně definujte povinnosti/zodpovědnost
  - měla by dělat jednu věc a dělat ji dobře
    - *Třída GrO představuje strukturu 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

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

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

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

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 („*Let q(x) be a property provable about objects x of type T. Then q(y) should be true for objects y of type S where S is a subtype of T.*“)
  - netýká se jen syntaxe, ale i sémantiky (tu je nutné vhodně popsat)
  - podtřída musí dodržovat kontrakt nadtřídy
- Vztah (ISA) musí být trvalý
- Příklad
  - Reálné číslo * komplexní číslo
  - Obdélník * kvádr

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

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.