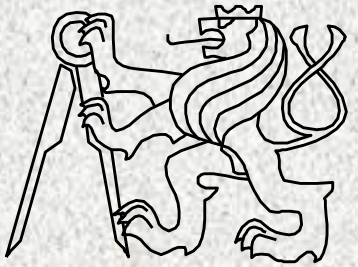
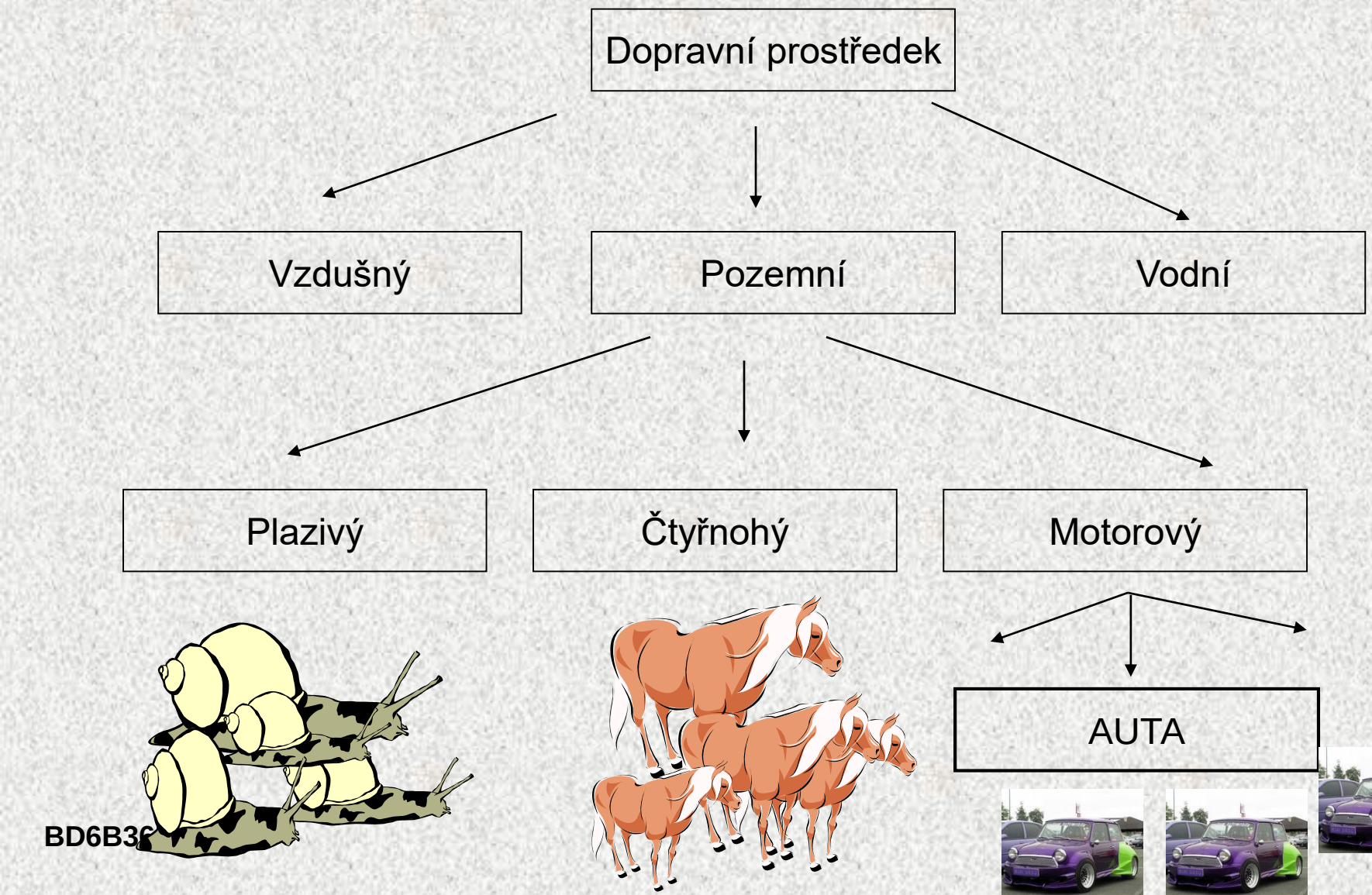
Třídy a dědičnost



BD6B36PJV

Fakulta elektrotechnická
České vysoké učení technické

Třídy a dědičnost



Dědičnost – základní vlastnosti

- Mechanismus umožňující
  - rozšiřovat datové položky tříd *(také modifikovat)*
  - rozšiřovat či modifikovat metody tříd
- Dědičnost umožní
  - vytvářet hierarchie tříd
  - „předávat“ datové položky a metody k rozšíření a úpravě
  - specializovat, „upřesňovat“ třídy
- Dvě základní výhody dědění
  - Dědění má praktický význam v znuvupoužitelnosti programového kódu
  - Dědičnost je základem polymorfismu

Dědičnost

Dědičnost:

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

Kompozice

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

Čítač jako datový typ

- Čítač zavedeme jako datový typ s operacemi

  zvetsit, *zmensit, nastavit* a *hodnota*

  a s datovými položkami

  hodn a *pocHodn*

- Grafické vyjádření:

- Poznámka: hodnota položky *pocHodn* bude stanovena při vytvoření objektu

Třída Citac

```
public class Citac {
  private int hodn;
  private int pocHodn;

  public Citac(int ph) {
     pocHodn = ph;
     hodn = ph;
  }

  public void zvetsit() {hodn++;}

  public void zmensit() {hodn--;}

  public void nastavit() {hodn = pocHodn;}

  public int hodnota() {return hodn;}
```

Použití čítače jako objektu

```
public static void main(String[] args) {
    int volba;
    Citac citac = new Citac(0);
//        Citac citac1 = new Citac(3);
    do {
      System.out.println("Hodnota = "+citac.hodnota());
      volba = menu();
      switch (volba) {
        case 1: citac.zvetsit(); break;
        case 2: citac.zmensit(); break;
        case 3: citac.nastavit(); break;
     //       case 4: System.out.println("nic"); break;
      }
    } while (volba>0);
    System.out.println("Konec");
  }
```

Třída CitacModulo, dědění

název typu

datové položky

operace (metody)

Třída CitacModulo, dědění

```
public class CitacModulo extends Citac {
public CitacModulo(int ph) {
super(ph);
}
@Override
public void zvetsit() {
System.out.println("  " + hodn % 5);
hodn++;
hodn = ((hodn % 5) == 0) ? pocHodn : hodn;
    }
}
```

Menu jako datový typ

- Menu zavedeme jako datový typ s operacemi
  vyber() a *volba()*
  a s datovými položkami
  prvky, *vyzva* a *volba*
- Grafické vyjádření:

Třída Menu

```
public class Menu {
  String[] prvky;
  String vyzva;
  private int volba;
  public Menu(String[] prvky, String vyzva) {
    this.prvky = prvky;
    this.vyzva = vyzva;}
  public int vyber() {
      Scanner sc = new Scanner(System.in);
      do {
      for (int i=0; i<prvky.length; i++)
        System.out.println(prvky[i]);
      System.out.print(vyzva);
      volba = sc.nextInt();
      if (volba<0 || volba>=prvky.length) {
        System.out.println("Nedovolená volba");
        volba = -1;
      }
    } while (volba<0);
    return volba;}
  public int volba() {
    return volba;
  }
```

Čítač a Menu jako objekt

```
public class Citac_Test_Objekt_Menu
public static void main(String[] args) {
    String[] nabidka = {
      "0) Konec", "1) Zvětšit", "2) Zmenšit", "3) Nastavit"
    };
    Citac citac = new Citac(0);
    Menu menu = new Menu(nabidka, "Vaše volba: ");
    do {
      System.out.println("Hodnota = "+citac.hodnota());
      switch (menu.vyber()) {
        case 1: citac.zvetsit(); break;
        case 2: citac.zmensit(); break;
        case 3: citac.nastavit(); break;
      }
    } while (menu.volba()!=0);
    System.out.println("Konec");
  }
}
```

Třída MenuX jako potomek třídy Menu - dědění

1. zdědí od třídy Menu:
   - všechny datové položky

     ```
     private String[] prvky;
     private String vyzva;
     private int volba;
     ```

   - všechny metody

     ```
     public int vyber();
     public int volba();
     ```

2. přidá
   - datovou položku
     - vytváření menu libovolné délky

       ```
       private int pocetPrvku, volny;
       ```

   - metody
     - přidávání položky menu `pridejPolozkuDoMenu(String s)`
     - zadání libovolné výzvy `pridejVyzvu(String s)`

Třída MenuX jako potomek třídy Menu

- vytváření menu libovolné délky **`(pocetPrvku + volny)`**
- přidávání jednotlivé položky menu **`(pridejPolozkuDoMenu(String s))`**
- zadání libovolné výzvy **`(pridejVyzvu(String s))`**

MenuX
prvky **vyzva** **volba** **pocetPrvku** **volny**
vyber() **volba()** **pridejPolozkuDoMenu(String s)** **pridejVyzvu(String s)**

Třída MenuX jako potomek Menu

```
class MenuX extends Menu{
//private String[] prvky;
//private String vyzva;
//private int volba;
// public int vyber() {
// public int volba() {
private int pocetPrvku;
private int volny=0;
 public MenuX(String[] prvky, String vyzva,int pocet) {
    super(prvky, vyzva);
    this.pocetPrvku = pocet;
}
public void pridejPolozkuDoMenu(String s){
   if (volny < pocetPrvku)prvky[volny++]=s;
  }
public void pridejVyzvu(String s){
    vyzva=s;
        }
}
```

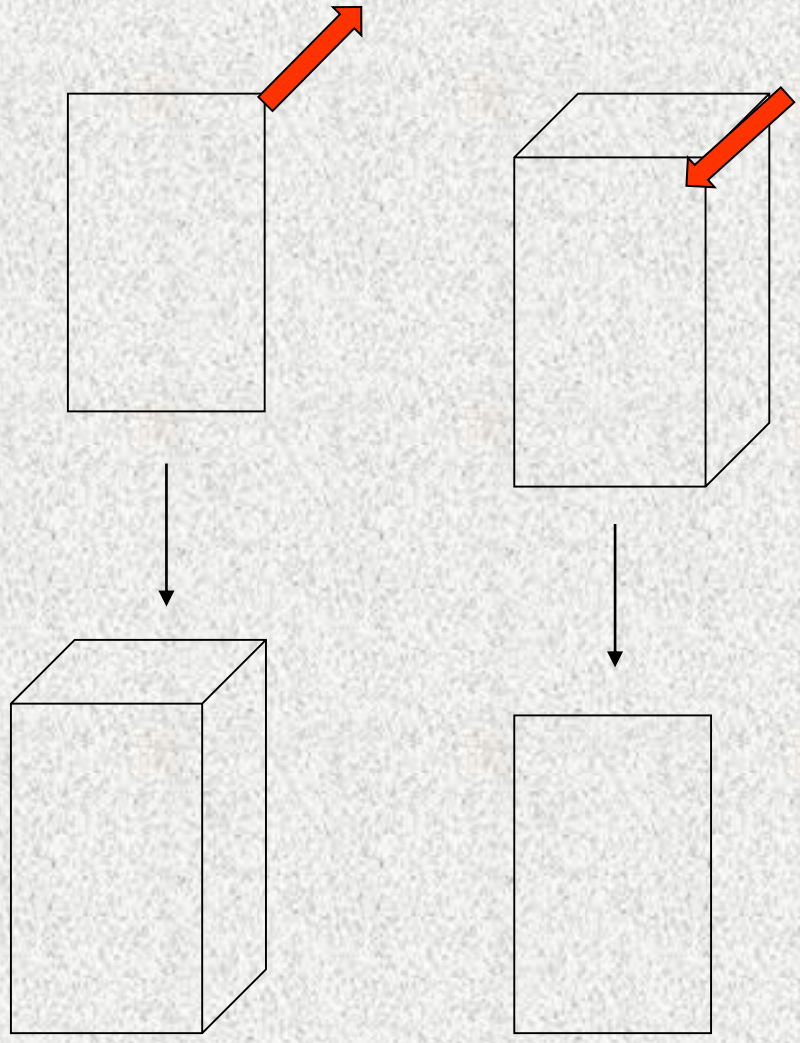
Třída Čítač TEST

```
public class Citac_Test_MenuX {
  public static void main(String[] args) {
    Citac citac = new Citac(0);
    String[] nabidka = new String[6];
    MenuX menu = new MenuX(nabidka, "Vaše volba: ", 6);
    menu.pridejPolozkuDoMenu("0) Konec");
    menu.pridejPolozkuDoMenu("1) Zvětšit");
    menu.pridejPolozkuDoMenu("2) Zmenšit");
    menu.pridejPolozkuDoMenu("3) Nastavit");
    menu.pridejPolozkuDoMenu("4) nove");
    menu.pridejPolozkuDoMenu("5) XXX");
    menu.pridejVyzvu("Vaše volba: ");
    do {
      System.out.println("Hodnota = "+citac.hodnota());
      switch (menu.vyber()) {
        case 1: citac.zvetsit(); break;
        case 2: citac.zmensit(); break;
        case 3: citac.nastavit(); break;
        case 4: System.out.println("nic"); break;
      }
    } while (menu.volba()!=0);
    System.out.println("Konec");
```

Je opravdu určení dědičnosti jednoduché?

Příklad:

- Je kvádr je rozšířený obdélník?
  - **IS A box extended rectangle?**
- Je obdélník zúžený kvádr?
  - **IS A rectangle narrowed block?**

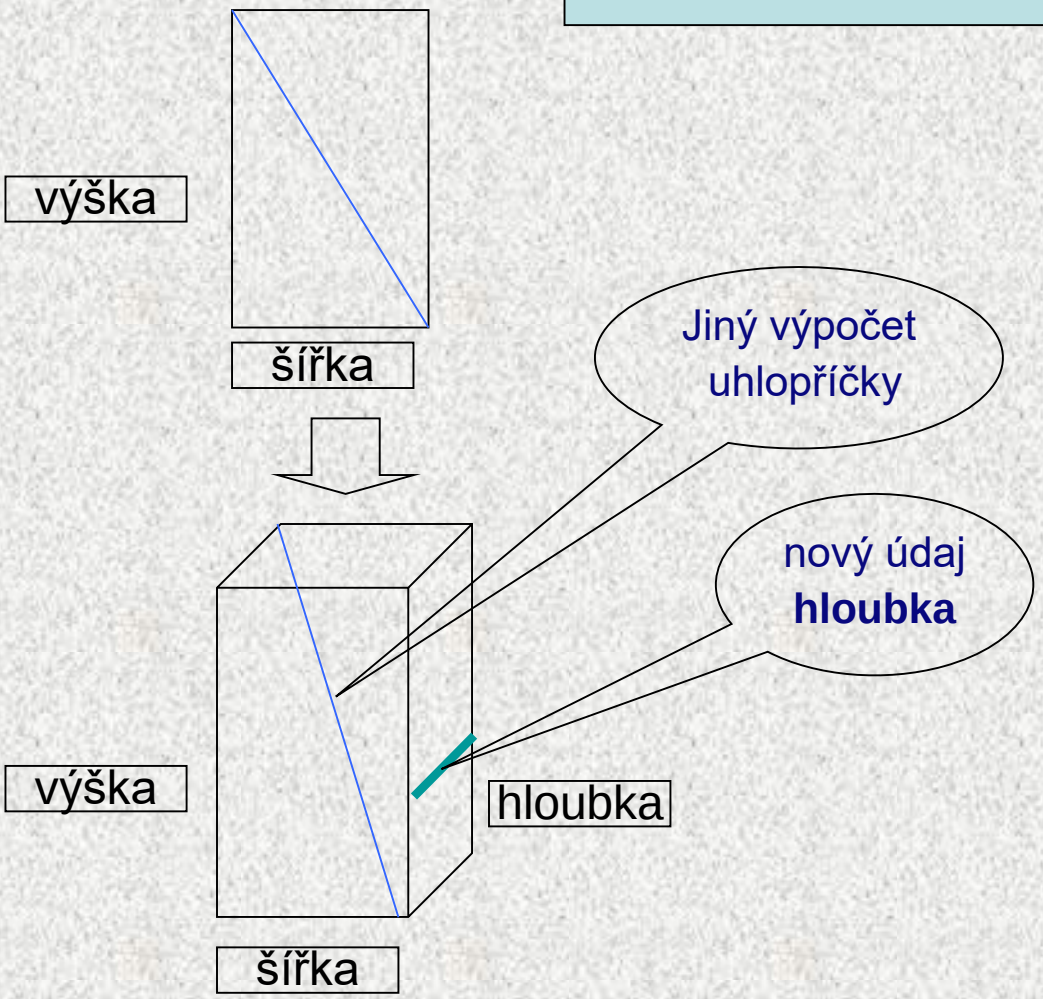


Dědičnost – Kvádr je rozšířený obdélník?

Obdelnik
sirka;
vyska;
delkaUhlopricky()
hodnotaSirky()

Kvadr extends Obdelnik
hloubka
delkaUhlopricky()

Kvádr je „rozšířením“ obdélníka?



Dědičnost – Kvádr je rozšířený obdélník?

```
public class Obdelnik1 {
  public int sirka;
  public int vyska;
  public Obdelnik1(int sirka, int vyska) {
    this.sirka = sirka;
    this.vyska = vyska;
  }
  public double delkaUhlopricky() {
    double pom;
    pom = (sirka * sirka) + (vyska * vyska);
    return Math.sqrt(pom);
  }
  public int hodnotaSirky() {
    return sirka;
  }
}
```

Dědičnost – Kvádr je rozšířený obdélník?

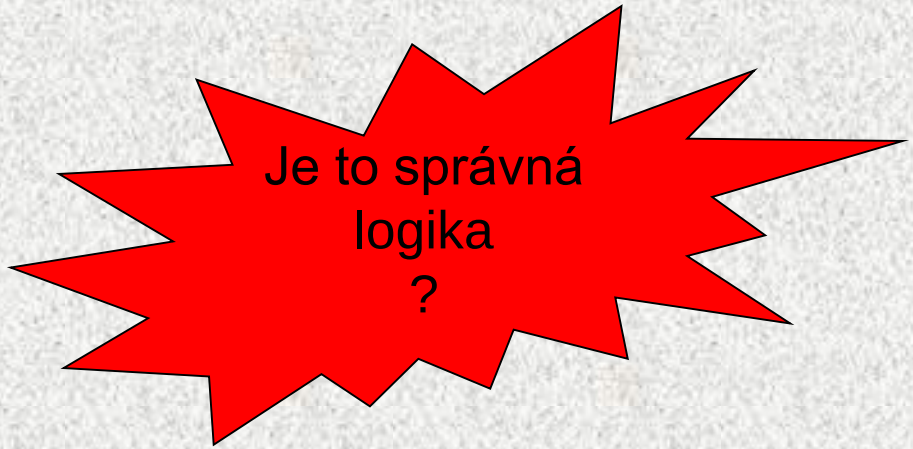
```
class Kvadr1 extends Obdelnik1 {
public int hloubka;

public Kvadr1(int sirka, int vyska, int hloubka) {
super(sirka, vyska); // volání konstruktoru předka
this.hloubka = hloubka;
}

public double delkaUhlopricky() {
                        // překrytí metody předka
double pom = super.delkaUhlopricky();
                        // volání metody předka
pom = (pom * pom) + (hloubka * hloubka);
return Math.sqrt(pom);
  }
}
```

Dědičnost - komentář

- Kvádr je rozšířením obdélníka o hloubku
- Potomek se deklaruje pomocí klauzule **`extends`**
  - **`Kvadr`** převezme proměnné **`sirka`** a **`vyska`**, metodu **`hodnotaSirky`**
  - Konstruktor se nedědí, ale může být v podtřídě využit, je volán slovem **`super`**
    - jako první příkaz v podřízeném konstruktoru, s parametry rodiče!
  - Není-li konstruktor volán přímo, je volán konstruktor bez parametrů!!!
    - musí tedy existovat, ať už implicitní či uživatelský!!!
- Potomek doplňuje novou proměnou **`hloubka`** a mění metodu **`delkaUhlopricky`**
- Objekty **`Kvadr`** mohou využívat proměnné **`sirka`**, **`vyska`** a **`hloubka`**, metody **`hodnotaSirky`** a vlastní **`delkaUhlopricky`**
- Metoda **`delkaUhlopricky`** třídy **`Kvadr`** překrývá (overriding, zastínění) metodu téhož jména v rodičovské třídě
- Jsou-li jiné parametry (jiný počet, jiný typ), jedná se o tzv. přetížení - overloading) – jedná se tedy o jinou, novou metodu!!

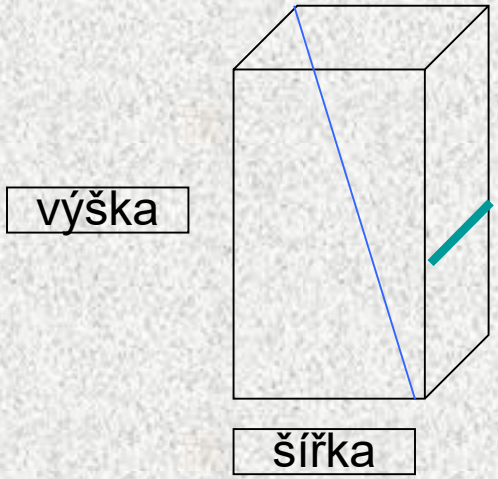
Dědičnost – Kvádr je rozšířený obdélník?

- Plocha vepsané elipsy?
- Jaký je obvod kvádru?
- ….



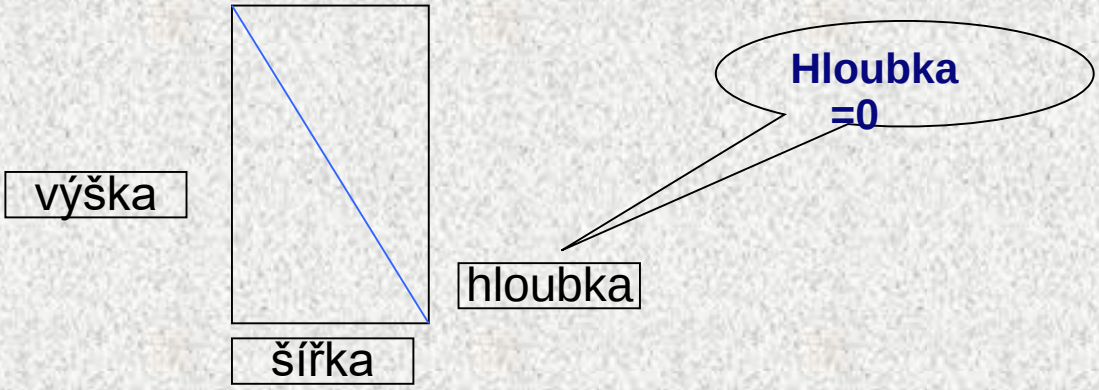
Dědičnost – Obdélník je speciální kvádr?

Kvadr
sirka;
vyska;
hloubka
delkaUhlopricky()
hodnotaSirky()



Obdélník je kvádr
s nulovou hloubkou?

Obdelnik extends Kvadr



Dědičnost – Obdélník je speciální kvádr?

```
public class Kvadr2{
public int sirka;
public int vyska;
public int hloubka;

public Kvadr2(int sirka, int vyska, int hloubka) {
this.hloubka = hloubka;
this.sirka = sirka;
this.vyska = vyska;
}

public int hodnotaSirky() {
    return sirka;
  }

public double delkaUhlopricky() {
double pom = (sirka * sirka) + (vyska*vyska) + (hloubka *
   hloubka);
return Math.sqrt(pom);
  }
}
```

Dědičnost – Obdélník je speciální kvádr?

```
class Obdelnik2 extends Kvadr2{

public Obdelnik2(int sirka, int vyska) {
super(sirka, vyska, 0);
}
}
```

Dědičnost – příklad

```
Obdelnik1 obd = new Obdelnik1(6, 8);
Kvadr1 kva = new Kvadr1(6, 8, 10);
System.out.println("Kvadr je rozsirenim obdelnika");
System.ouintln("Uhlopricka obdelnik:„+obd.delkaUhlopricky());
System.ouintln("Uhlopricka kvadr: " + kva.delkaUhlopricky());
System.out.println("Sirka kvadru je: " + kva.hodnotaSirky());
System.outintln("Sirka obdelnika  je: "+ obd.hodnotaSirky());
System.out.println("hloubka kvadru je: " + kva.hloubka);
//System.out.println("hloubka obdelnika je: " + obd.hloubka);
```

```
Kvadr je rozsirenim obdelnika
Uhlopricka obdelnik: 10.0
Uhlopricka kvadr: 14.142135623730951
Sirka kvadru je: 6
Sirka obdelnika  je: 6
hloubka kvadru je: 10
```

Dědičnost – příklad

```
Obdelnik2 obd2 = new Obdelnik2(6, 8);
Kvadr2 kva2 = new Kvadr2(6, 8, 10);
System.out.println("Obdelnik je specilani pripad kvadru");
System.ontln("Uhlopricka obdelnik: „+obd2.delkaUhlopricky());
System.outln("Uhlopricka kvadr: " + kva2.delkaUhlopricky());
System.outln("Sirka kvadru je: " + kva2.hodnotaSirky());
System.outln("Sirka obdelnika  je: " + obd2.hodnotaSirky());
System.out.println("hloubka kvadru je: " + kva2.hloubka);
System.out.println("hloubka obdelnika je: " + obd2.hloubka);
```

```
Obdelnik je specilani pripad kvadru
Uhlopricka obdelnik: 10.0
Uhlopricka kvadr: 14.142135623730951
Sirka kvadru je: 6
Sirka obdelnika  je: 6
hloubka kvadru je: 10
hloubka obdelnika je: 0
```

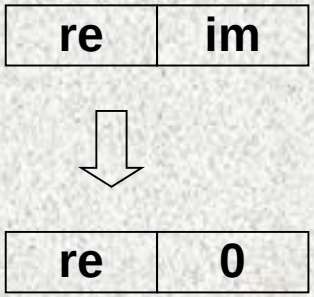
Dědičnost - komentář

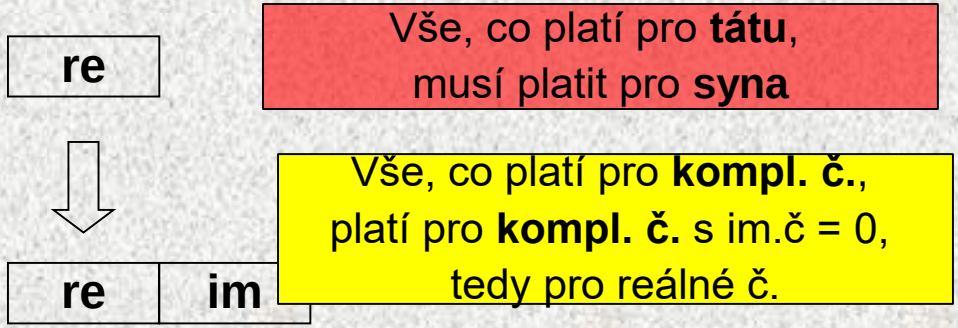
- Obdélník je „kvádrem“ s nulovou hloubkou
- Potomek se deklaruje pomocí klauzule **`extends`**
  - **`Obdelnik`** převezme proměnné **`sirka, vyska, hloubka`** metody **`hodnotaSirky, delkaUhlopricky`**
  - Konstruktor se dědí, parametr **`hloubka`** se nastaví do nuly
- Objekty **`Obdelnik`** mohou využívat proměnné **`sirka, vyska`** a **`hloubka,`** metody **`hodnotaSirky`** a **`delkaUhlopricky`**
- Správná logika??
  - Objem obdélníka?
  - Otočení okolo hrany?

Je obdélník potomek kvádru či naopak?

1. Kvádr je potomek obdélníka
   - Logické přidání rozměru, ale metody platné pro obdélník nefungují pro kvádr („vepište elipsu do obdélníka“, „obsah obdélníka“,..)
2. Obdélník je potomek kvádru
   - Logicky správná úvaha o „specializaci“ („vše co funguje pro kvádr, funguje i kvádr s nulovou hloubkou“), ale neefektivní implemetace (každý obdélník je reprezentován 3 rozměry)

- Odpověď 2 je správná?
  - A co objem obdélníka?
- Podobná situace: Komplexní číslo vs. Reálné číslo





Předek či potomek, vztah „ISA“

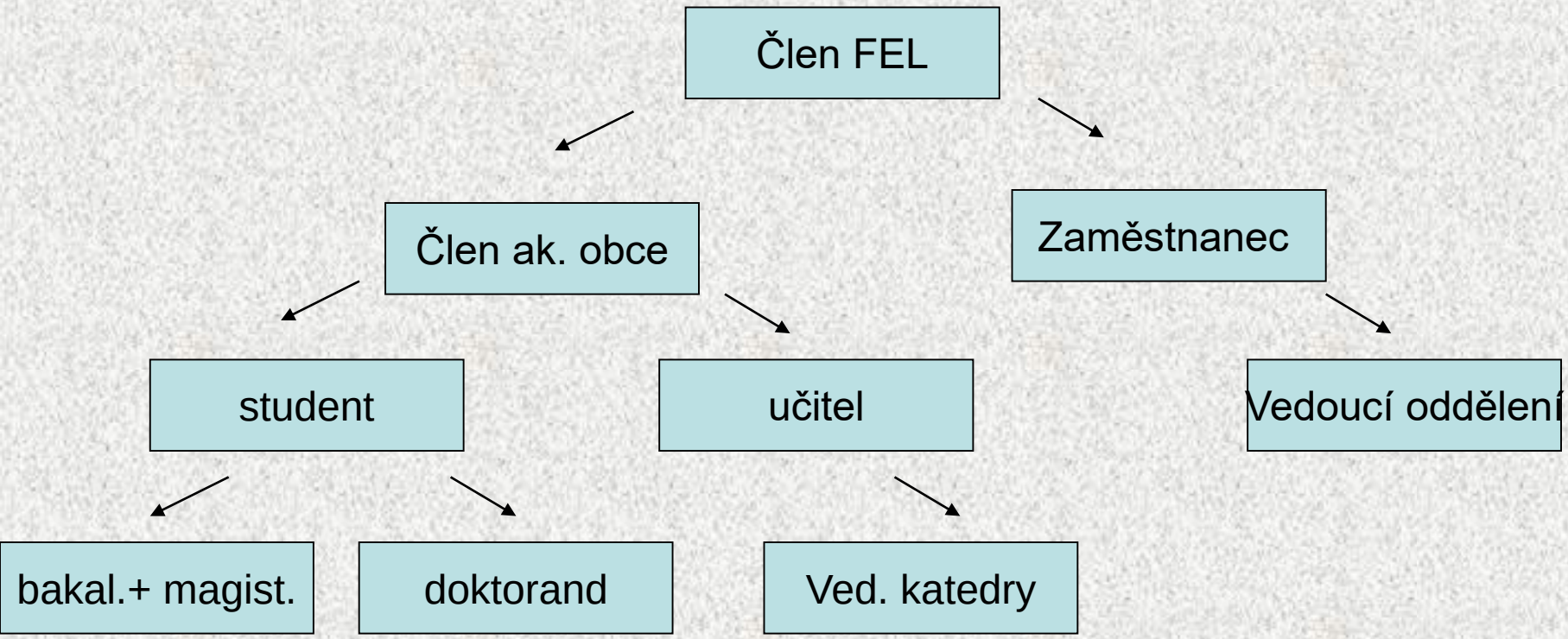
- Je Usecka potomkem Bodu?
  + Usecka rozšiřuje Bod o jeden jeho konec
  - Usecka nevyužije ani jednu metodu Bodu
  - ISA vztah: ? Usecka je Bod? – **NE** → Usecka není potomkem Bodu

- 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

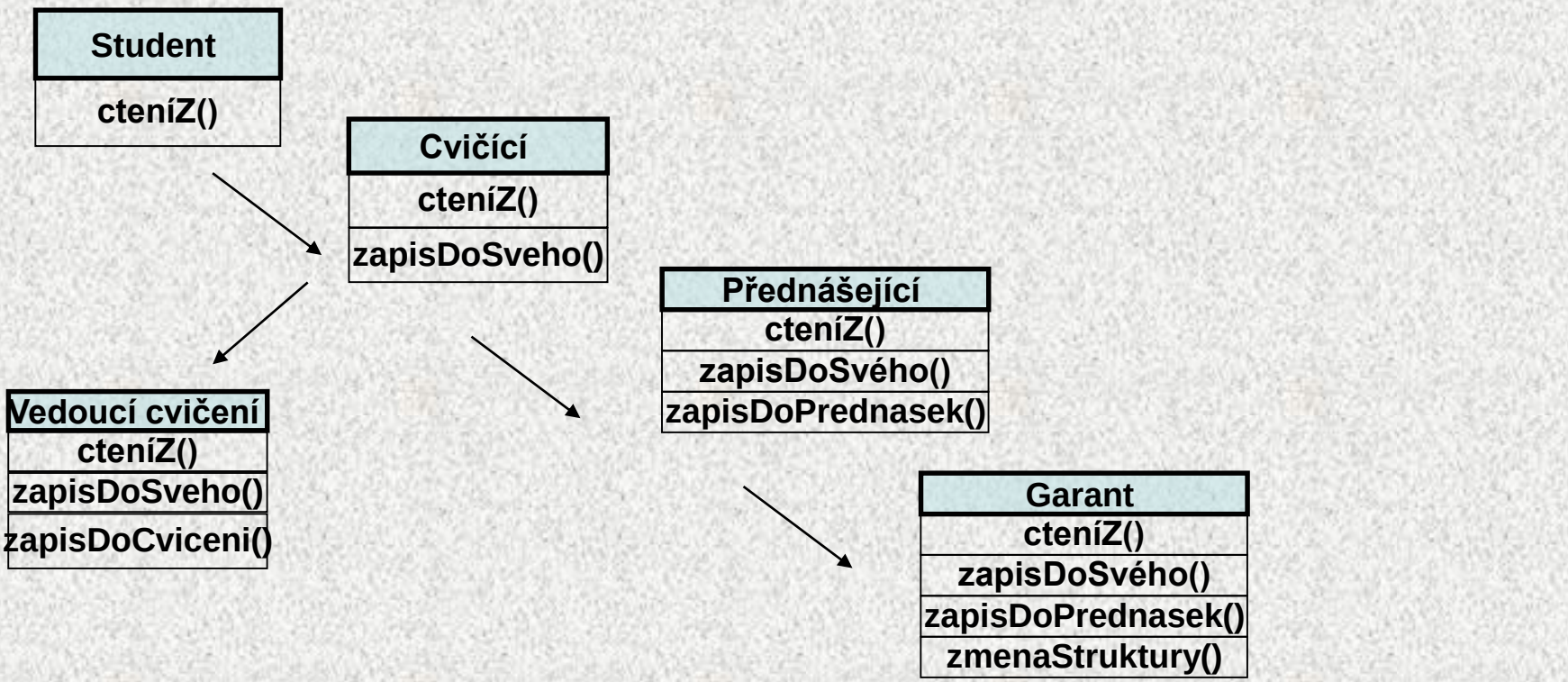
Další zamyšlení

Příklad vztahu dědičnosti - zaměstnanci



- Zde funguje ISA na 100%

Příklad vztahu dědičnosti - Přístupová práva



- Zde je jasná dědičnost, ale ne ISA
  - Je to model přístupu k EDUXu ☺

Dědičnost – shrnutí

Vlastnosti dědění v Javě:

- opakovatelné využití programových částí (atributů (členských proměněných) a metod)
- **možnosti**
  - **využití atributů či metod rodiče,**
  - **upřesnění metod (i atributů) modifikací**
  - **přidání dalších atributů a metod**
- vytváření hierarchie objektů, postupně více specializovaných
  - specializace chování objektů !
  - existuje kompatibilita děděných objektů směrem "nahoru"

Poznámky:

- Východisko polymorfismu
- Jazyk Java zná dědění od jediného předka!
- potřeba zdědit vlastnosti od více předků se řeší rozhraním (`interface`) – viz další přednášky a úvod ve cvičeních

Hierarchie tříd, definice

- Třída *Tpod,* která je podtřídou třídy *Tnad,* dědí vlastnosti nadtřídy *Tnad* a rozšiřuje je o nové vlastnosti; některé zděděné vlastnosti mohou být v podtřídě modifikovány

- Pro instanční metody to znamená:
  - každá metoda třídy *Tnad* je i metodou třídy *Tpod,* v podtřídě však může mít jinou implementaci (může být zastíněna - override)
  - v podtřídě mohou být definovány nové metody

- Pro strukturu objektu to znamená:
  - instance třídy *Tpod* mají všechny členy třídy *Tnad* a případně další

Hierarchie tříd v dokumentaci

- V on-line dokumentaci jazyka Java týkající se třídy *ArrayList* najdeme následující obrázek:

```
java.lang.Object
    |
    +--java.util.AbstractCollection
        |
        +--java.util.AbstractList
            |
            +--java.util.ArrayList
```

- Tento obrázek vyjadřuje, že:
  - třída *ArrayList* (definovaná v balíku *java.util*) je podtřídou třídy *AbstractList*
  - třída *AbstractList* je podtřídou třídy *AbstractCollection*
  - třída *AbstractCollection* je podtřídou nejvyšší třídy java.lang.*Object*

poznámka

Třída Object

Pokud v definici třídy není explicitně uvedena nadtřída, je jako nadtřída automaticky použita třída **`Object`**

`class X {}` je ekvivalentní s **`class X extends Object {}`**

- metody z třídy **`Object`** jsou tedy zděděny ve všech třídách (definovaná v **`java.lang`**)
- třída **`Object`**, implementuje několik základních metod :

```
public boolean equals(Object o);
      // porovnává objekt s parametrem
 public String toString();
      // vrací textovou informaci o objektu
 public int hashCode();
      // vrací jednoznačný hešovací kód objektu
 protected Object clone();
      // vytváří kopii objektu
      // a další ….
…
```

Metoda `toString`

```
public String toString(){
    return getClass().getName() + "@" +
                    Integer.toHexString(hashCode());
}
```

- výsledkem volání `x.toString( )` je řetěz znakové reprezentace objektu `x`,
- metodou je zavedena implicitní typová konverze z typu objektu *x* na řetězec, použitá např. při výpisu objektu

Příklad zastínění

```
public String toString(){
  return ("Obdelnik: "+ sirka + " x "+ vyska);
}
```

Poznámka: generuje také NetBeans

Zastínění metod `equals`

```
public boolean equals(Object obj) {
  return (this == obj);
}
```

- standardní chování neporovnává položky, ale reference

Příklad zastínění

```
public boolean equals(Object o){
  if(!(o instanceof Obdelnik))return false;
                              // porovnam jen s obdelniky
  Obdelnik obd = (Obdelnik) o;  //pretypovani
  return (sirka == obd.sirka)&&(obd.vyska==vyska);
 }
```

Poznámka: generuje také NetBeans

Příklad vlastností metody **equals** a **toString**

```
public class ObdelnikTest {
    public static void main(String[] args) {
    Obdelnik prvni = new Obdelnik(5,7);
    Obdelnik druhy = new Obdelnik(5,7);
    System.out.println("Prvni: " + prvni);
    System.out.println(„Druhy: " + druhy);
    System.out.println("Stejne? "+(prvni==druhy));
    System.out.println("Stejne?"+prvni.equals(druhy));
    }
}
```

Standardní funkce
Prvni: pjv2.Obdelnik@70dea4e Druhy: pjv2.Obdelnik@5c647e05 Stejne ==? false Stejne equals? false

Zastíněné funkce
Prvni: [5,7] Druhy: [5,7] Stejne ==? false Stejne equals? true

Metody **`equals()`** a **`hashCode()`**

- Jestiže se překrývá metoda **`equals`**, pak je nutné překrýt i metodu **`hashCode()`**, **důvody u kolekci**
- Jestliže jsou dva objekty shodné, je třeba, aby generovaly tentýž **`hashcode()`**
- Generuje NetBeans ☺

```
public int hashCode() {
        int hash = 7;
        hash = 59 * hash + this.sirka;
        hash = 59 * hash + this.vyska;
        return hash;
    }
}
```

Dědění - operátor **`super`**

- Pokud je potřeba zavolat ve třídě metodu či konstruktor z nadtřídy, která/ý byl/a přepsán/a je k dispozici operátor **`super`**:

```
class Rodic {
  public int i;
  public Rodic(int parI) { i = parI; }

public class Potomek extends Rodic {
  public Potomek() {          // musí být! - proč?
    super(8);  }

public String toString()
  {return "Potomek";}

  public static void main(String[] args) {
    Potomek pot = new Potomek();
    System.out.println("i= " + pot.i + "   " + pot);}}
```

```
i= 8   Potomek
```

Dědění - operátor **super**

```
class Bod2X {
  double x, y;
  BodX(){x=7;y=8;}
  BodX(double x){ this.x = x; y = 0;}
  double radius() { return Math.sqrt(x*x + y*y); }
}
public class B3D extends Bod2X {
  double z;
  B3D(){super();this.z=199;}
    @Override
  double radius() {
    double rad2D = super.radius();
    return Math.sqrt(rad2D * rad2D + z*z);
  }
```

Dědění - operátor **super**

```
public static void main(String[] args) {
     BodX bX = new BodX();
    System.out.println("BodX, radius "+ bX.radius());
      bX = new B3D();
    System.out.println("BodX, radius "+ bX.radius());
    B3D  b3X= new B3D();
    System.out.println("Bod3D, radius "+ b3X.radius());
  }
}
```

```
BodX, radius 10.630145812734650
BodX, radius 199.28371734790576
Bod3D, radius 199.28371734790576
```

Hierarchie tříd, vlastnosti

- Pro referenční proměnné to znamená:
  - proměnné typu *Tnad* může být přiřazena reference na objekt typu *Tpod*
  - na objekt referencovaný proměnnou typu *Tnad* lze vyvolat pouze metodu deklarovanou ve třídě *Tnad*; jde-li však *o* objekt typu *Tpod,* metoda se provede tak, jak je dáno třídou *Tpod,* je-li předefinována, jinak se provede metoda deklarovaná ve třídě *Tnad*
  - hodnotu referenční proměnné typu *Tnad* lze přiřadit referenční proměnné typu *Tpod* s použitím přetypování, které zkontroluje, zda referencovaný objekt je typu *Tpod*

- Vztah *nadtřída – podtřída* je tranzitivní
  - jestliže je x nad třídou y a y je nad třídou z, pak je x nadtřídou z

Kompatibilita objektů při dědění

```
public class Bod2X {
    public double x, y;
    public Bod2X(double x, double y) {
        this.x = x;
        this.y = y;
    }
    public Bod2X(double x) {
        this.x = x;
        y = 0;
    }
    public double radius() {
        return Math.sqrt(x * x + y * y);
    }
}
```

```
public class Bod3X extends Bod2X {
    public double z;
    public Bod3X(double x, double y, double z) {
        super(x,y);
        this.z = z;
    }
    @Override
    public double radius() {
        double rad2D = super.radius();
        return Math.sqrt(rad2D * rad2D + z * z);
    }
}
```

Kompatibilita objektů při dědění

```
Bod2X b2X;
Bod3X b3X;
b2X = new Bod2X(1, 2);
System.out.println("B2X, radius " + b2X.radius());
b3X = new Bod3X(1, 2, 3);
System.out.println("B3X, radius " + b3X.radius());
b2X = new Bod3X(1, 2, 3);
System.out.println("B2X=Bod3X, radius " + b2X.radius());
//      b3X = new Bod2X(1, 2);
b3X = new Bod3X(1, 2, 3);
b2X = b3X;
System.out.println("B3X, radius " + b2X.radius());
System.out.println("B2X=B3X, radius " + b2X.radius());
b2X = new Bod2X(1, 2);
// b3X=b2X;
b3X = (Bod3X) b2X;
```

Primitivní typy jako objekty

- Do kolekcí lze vkládat pouze reference na objekty, nikoli primitivní typy (které primitivní typy v Javě máme?) – platilo do Java 1.5
- Primitivní typ např. čísla typu *int*, musíme je nejprve zabalit (wrap) do objektů typu java.lang.*Integer*

```
ArrayList <Integer> cisla = new ArrayList <Integer>();
cisla.add( new Integer(10));
cisla.add( new Integer(20));
System.out.println(cisla.get(0));      // vypíše se 10
System.out.println(cisla.get(1));     //  vypíše se 20
Integer prvni = (Integer)cisla.get(0);
```

- Číslo, které je v objektu typu **`Integer`** uloženo, získáme metodou **`intValue:`** `int n = prvni.intValue();`
- Podobné obalovací třídy (wrapper classes) jsou v Javě definovány pro všechny (celkem 8) primitivní typy.

Kompozice objektů

- Obsahuje-li deklarace třídy členské proměnné objektového typu, pak mluvíme o kompozici objektů
- Kompozice vytváří hierarchii objektů, nikoli však dědičnost
- Struktura **„HAS“**

Příklad:

Každá osoba je charakterizovaná těmito atributy (třída Osoba):

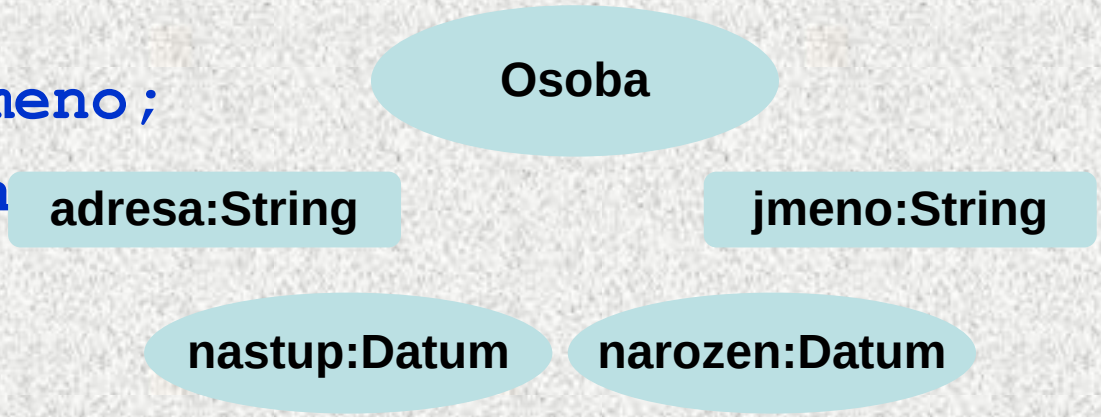
- Jménem
- Adresou
- Datum narození
- Datum nástupu

Datum je charakterizováno těmito atributy(třída Datum):

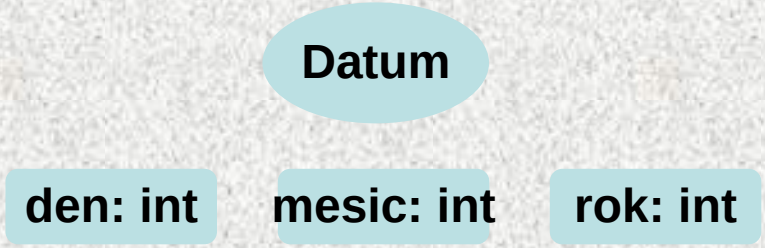
- Den
- Měsíc
- Rok

Kompozice objektů - příklad

```
class Osoba {
String adresa, jmeno;
Datum narozen, na
…}
```



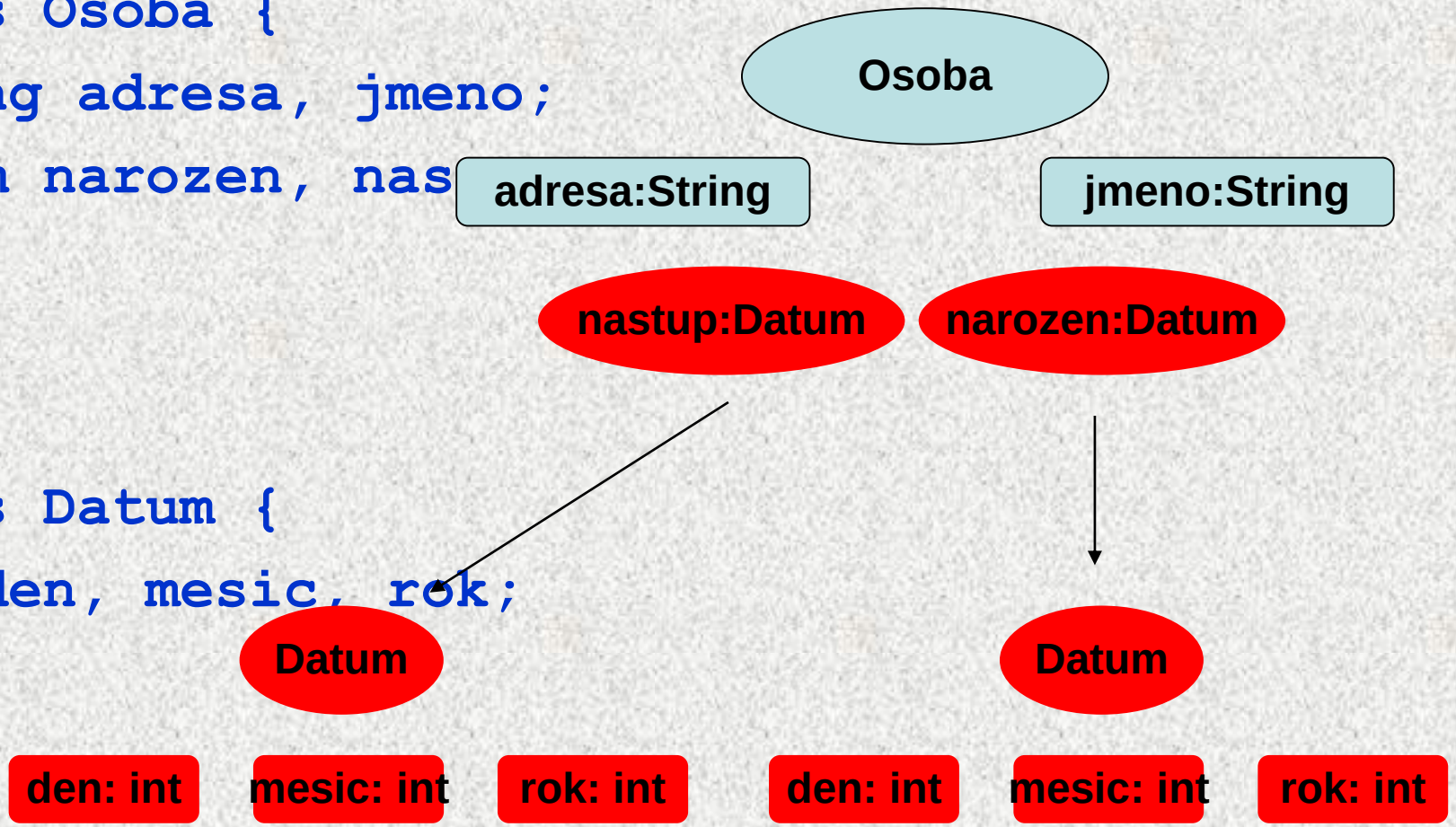

```
class Datum {
int den, mesic, rok;
…}
```



Kompozice objektů - příklad

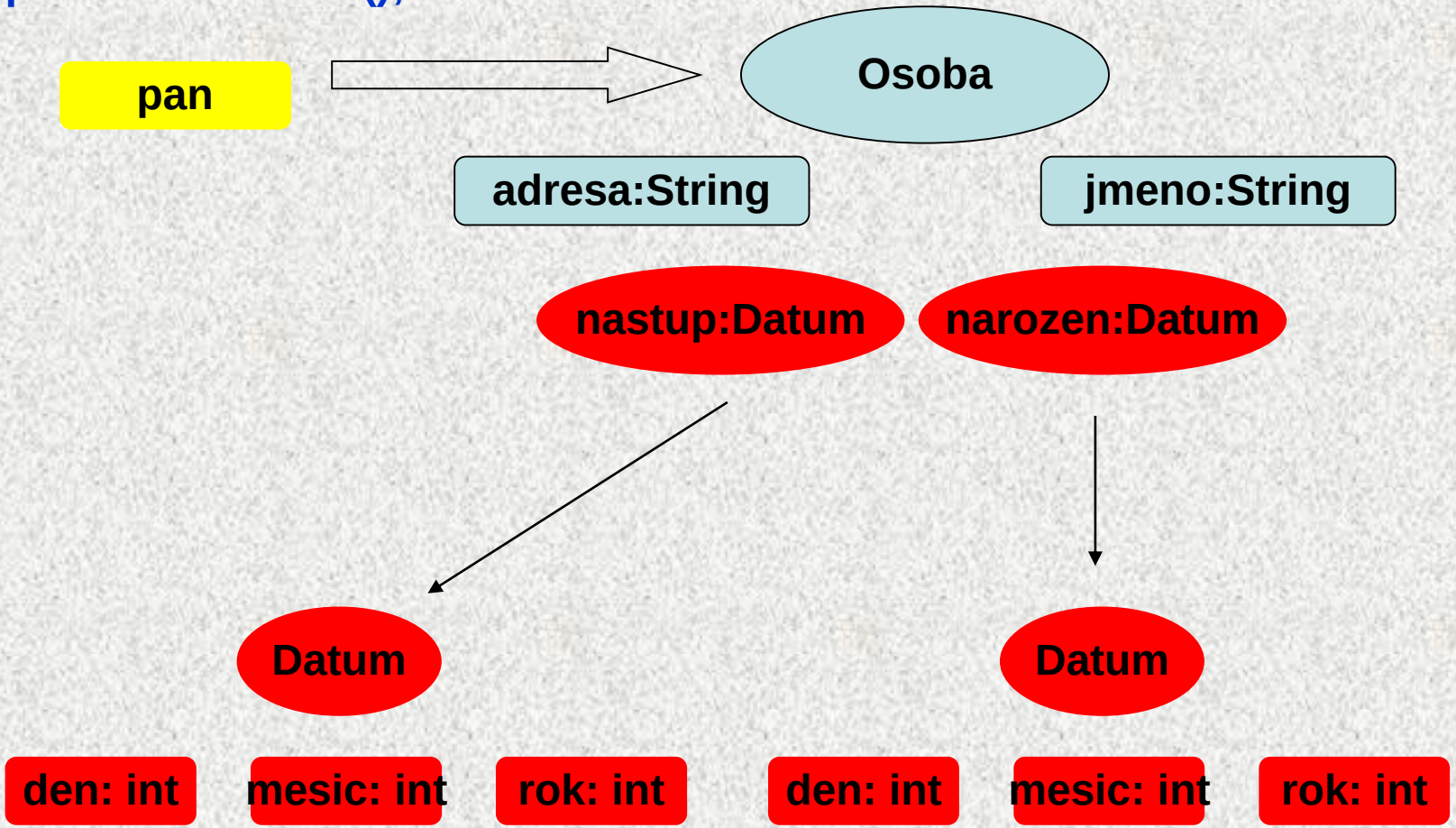
```
class Osoba {
String adresa, jmeno;
Datum narozen, nas
…}
```

```
class Datum {
int den, mesic, rok;
…}
```



Kompozice objektů - příklad

Osoba pan = new Osoba();



Kompozice objektů – `class Datum`

```
class Datum {
private int den, mesic, rok;
  public Datum (int den, int mesic, int rok){
   this.den=den;
   this.mesic=mesic;
   this.rok=rok;
}}

public int getMesic () {
return mesic;
}
public void putMesic (int mesic) {
test();
this.mesic = mesic;
}
private void test (){
if (this.mesic > 12) {
System.out.print("\nnedovolený mesic " + mesic + "nastavuji 12");
this.mesic=12;
}
```

Kompozice objektů – `class Osoba`

```
class Osoba {
String adresa, jmeno;
public  Datum narozen, nastup;
public Osoba (Datum narozen, Datum nastup, String jmeno,
   String adresa){
 this.narozen = narozen;
 this.nastup = nastup;
 this.adresa = adresa;
 this.jmeno = jmeno;
}
```

Kompozice objektů - příklad

```
public class Kompozice{
public static void main(String[] args){
Datum narozen = new Datum(12,3,1444);
Datum nastup = new Datum(3,10,2222);
Osoba pan= new Osoba(narozen, nastup, "Ryba", "rybník");
…
System.out.print(pan.narozen.getMesic());
…
if (pan.narozen.getMesic() < 12) {
   narozen.putMesic(pan.narozen.getMesic()+1);}
       else {
         pan.narozen.putMesic(1);
}
```

Dědičnost x kompozice - příklad

Příklad:

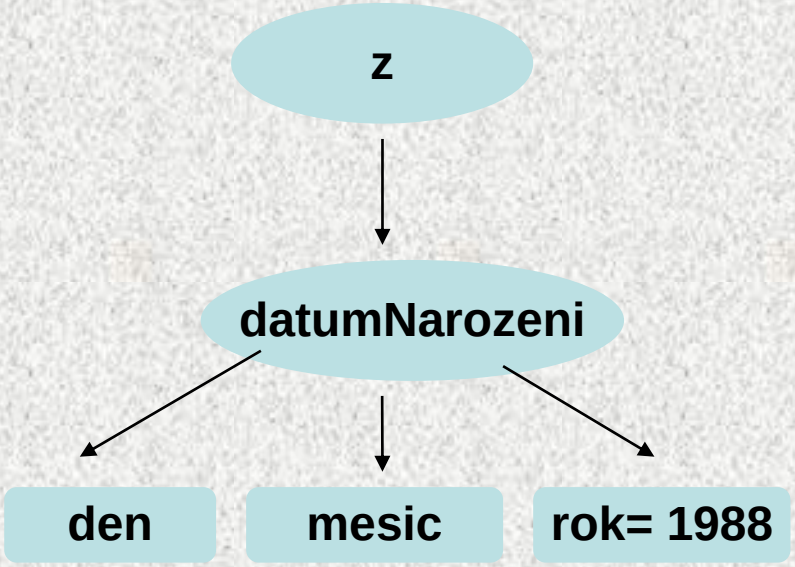
- Každý ředitel „**má**“ datum narození **(HAS)**, ale „**je**“ zaměstnancem **(ISA)**

```
class Zamestnanec {
Datum datumNarozeni;  // HAS
datumNarozeni= new Datum (21, 5, 2000);
…}

class Reditel extends Zamestnanec{     // ISA
Datum datumPovyseni;// HAS
datumPovyseni= new Datum (1, 1, 2013);
…}
```

- **Zamestnanec má (HAS)** datum narození
- **Reditel má (HAS)** datum narození od **Zamestnanec a má (HAS)** datum povýšení vlastní
- **Reditel je (ISA) Zamestnanec**

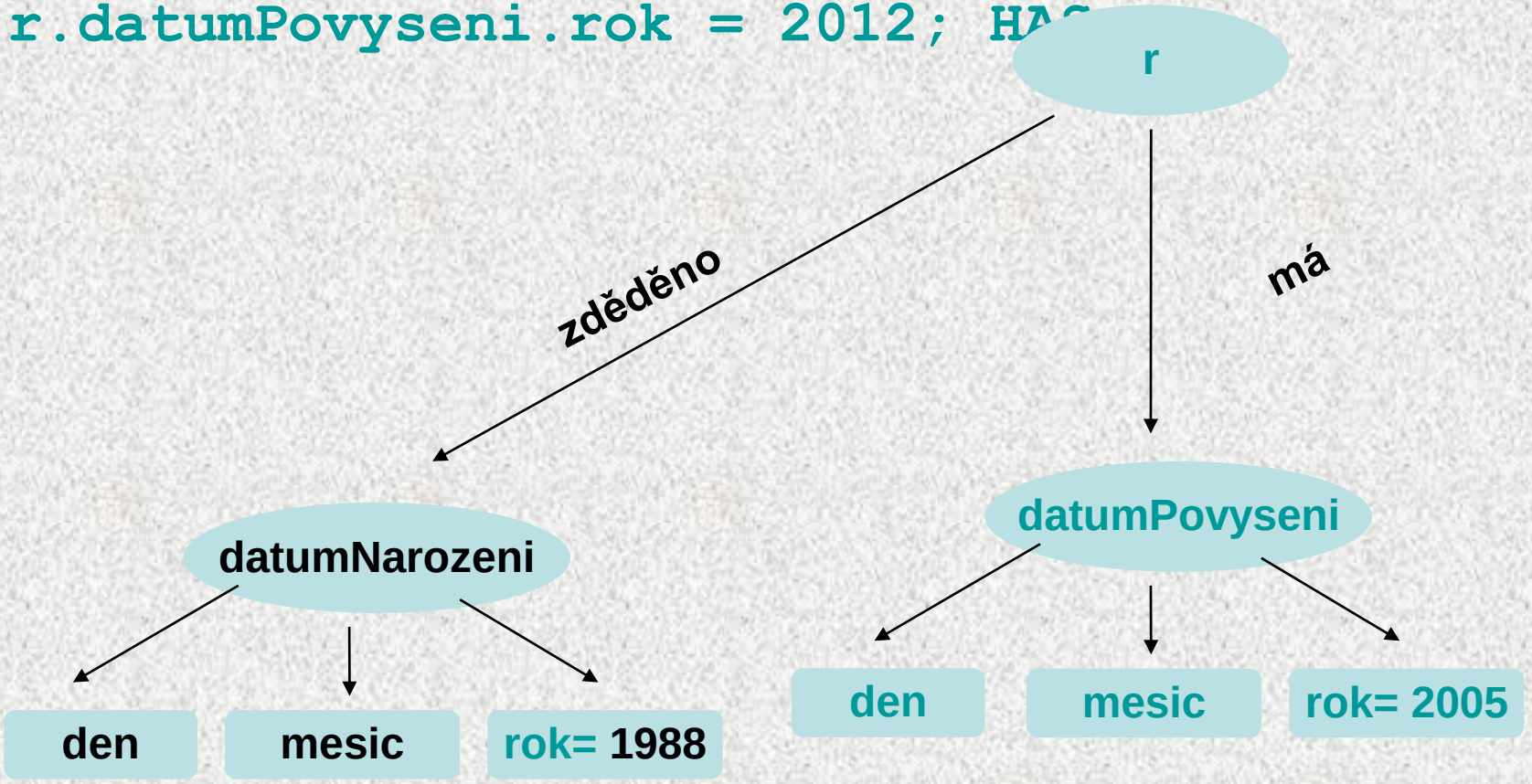
Dědičnost x kompozice - příklad

```
Zamestnanec z = new Zamestnanec();
z.datumNarozeni.rok = 2001;
```



Dědičnost x kompozice - příklad

```
Reditel r = new Reditel();
r.datumNarozeni.rok = 1978; HAS dědí
r.datumPovyseni.rok = 2012; HA
```



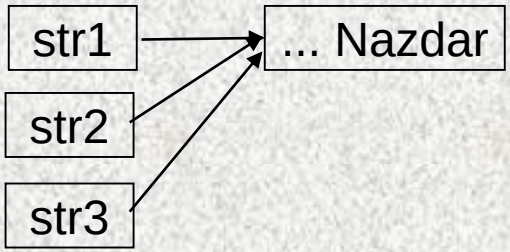
Dědičnost x kompozice

- **Základní vlastnosti dědění objektů:**
  - vytváření odvozené třídy (potomka, dceřinou třídu, podtřídu) pomocí **`extend`**
  - odvozená třída je specializovanější
    - přidané proměnné (překrytí proměnné)
    - přidané či modifikované metody
    - přístup k proměnným a metodám rodiče (supertřídě, předkovi, bázové třídě)
- **Kompozice objektů je tvořena atributy objektového typu, pouze je skládá**
- Rozlišení mezi kompozicí či děděním:
  - pomůcka: „Je“ test - příznak dědění **(ISA)**,
    „Má“ test - příznak kompozice **(HAS)**,

Třída String

- Objekty knihovní třídy *java.lang.String* jsou řetězy znaků
- Od ostatních tříd se liší třemi specialitami:
  - objekt typu *String* lze vytvořit literálem (posloupnost znaků uzavřená mezi uvozovky, ”Pepa”, “Další řetězec”)
  - hodnotu objektu typu *String* nelze jakkoli změnit
  - operací konkatenace čili zřetězení je nejen realizována metodou **`concat`**, ale i přetíženým operátorem +
- Příklady referečních proměnných typu *String:*

```
String str1 = "Nazdar";
String str2 = str1;
String str3 = "Nazdar";
```



- Java si eviduje seznam všech vytvořených řetězců a pokud již stejný existuje, nevytváří jeho kopii

Důležitá třída

Operace s řetězy

- Spojení řetězců (konkatenace) +
  "abc" + "123" výsledek je "abc123"
- Jestliže jeden operand operátoru + je typu *String* a druhý je jiného typu, pak druhý operand se převede na typ *String* a výsledkem je konkatenace řetězů
  "abc" + 5 výsledek je "abc5"
  "a" + 1 + 2 výsledek je "a12"
  "a" + 1 + (-2) výsledek je "a1-2"
- Porovnávání řetězů
  - relační operátory == a != porovnávají reference, nikoliv obsah řetězů
  - pro porovnání řetězů na rovnost slouží metoda *equals*

```
String s1 = "abcd" ;
String s2 = "ab" ;
String s3 = s2 + "cd" ;
System.out.println(s1==s3); // vypíše false
System.out.println(s1.equals(s3));// vypíše true
```


Důležitá třída

Operace s řetězy - porovnávání

- Metoda *compareTo*:

```
String s = "abcd";
System.out.println(s1.compareTo("abdc"));//vypíše -1
System.out.println(s1.compareTo( "abcd"));//vypíše 0
System.out.println("abdc".compareTo(s1));// vypíše 1
String s = "nazdar";
int delka = s.length();              // délka je 6
char znak = s.charAt(1);             // znak je 'a'
String ss = s.substring(2,4);              // ss je "zd"
int z1 = s.indexOf('a');             // z1 je 1
int z2 = s.lastIndexOf('a');               // z2 je 4
int z3 = s.lastIndexOf('A');               // z3 je -1
```

- Hodnotu referenční proměnné typu *String* lze změnit (odkazuje pak na jiný řetěz), vlastní řetěz změnit nelze

Příklad - palindrom

- Napišme program, který přečte jeden řádek a zjistí, zda se po vynechání mezer jedná o palindrom (čte se stejně zpředu jako zezadu, např. “kobyla ma maly bok”)
- funkce s parametrem typu *String* a výsledkem typu *boolean*:

```
static boolean jePalindrom(String str) {
    int i = 0, j = str.length()-1;
    while (i<j) {
      while (str.charAt(i)==' ') i++;
      while (str.charAt(j)==' ') j--;
      if (str.charAt(i)!=str.charAt(j))return false;
      i++; j--;
    }
    return true;
}
```

Příklad - palindrom

```
public class Palindrom {
 public static void main(String[] args) {
 System.out.println( "Zadejte jeden řádek" );
 String radek = (new Scanner(System.in)).nextLine();
 String vysl;
 if (jePalindrom(radek)) vysl = "je" ;
    else vysl = "není" ;
    System.out.println("Na řádku"+vysl+ " palindrom"
  );
     }
 static boolean jePalindrom(String str) {
     ...
     }
}
```


Důležitá třída

Pole znaků a řetěz

- Příklad: funkce pro převod celého čísla na řetěz tvořený zápisem čísla v hexadecimální soustavě

```
final static String HEXA = "0123456789abcdef";
static String hexaToDecimal(int x) {
if (x==0) return "0" ;
char[ ] znaky = new char[9];
int y;
if (x<0) y = -x; else y = x;
int prvni = 9;
do {
 prvni--;
 znaky[prvni] = HEXA.charAt(y%16);
 y = y / 16;
} while (y>0);
if (x<0) {
 prvni--; znaky[prvni] = '-';
   }
return new String(znaky, prvni, 9-prvni);
}
```

65%16=1
65/16 = 4
4%16=4
4/16=0
>>>> 65 > 41

Pro zájemce