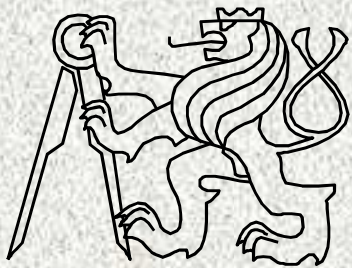


Třídy a dědičnost

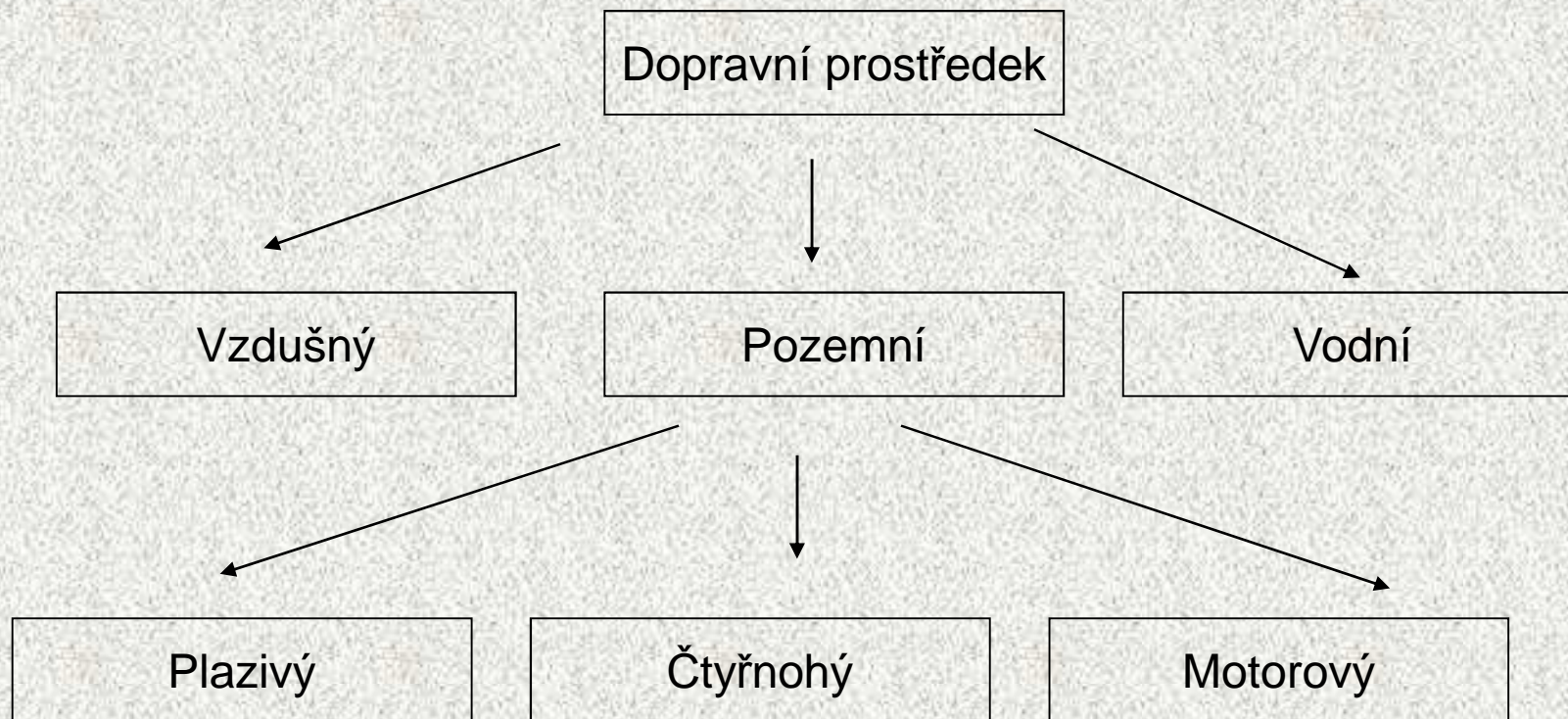


A0B36PR1-Programování 1
Fakulta elektrotechnická
České vysoké učení technické

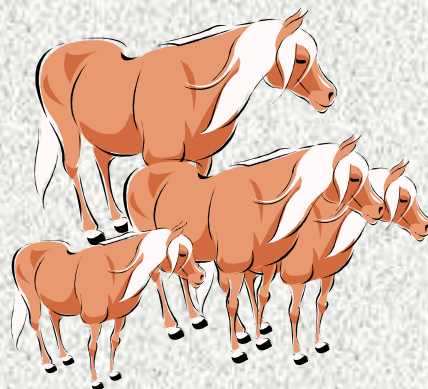
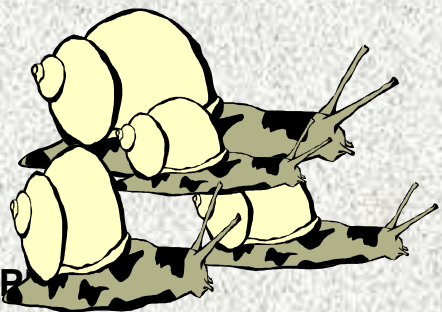
Dnešní témata

- Dědičnost
- Kompozice
- Typ `string`

Třídy a dědičnost



A0B36P



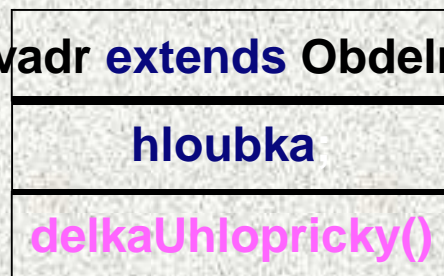
Dědičnost – základní vlastnosti

- 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

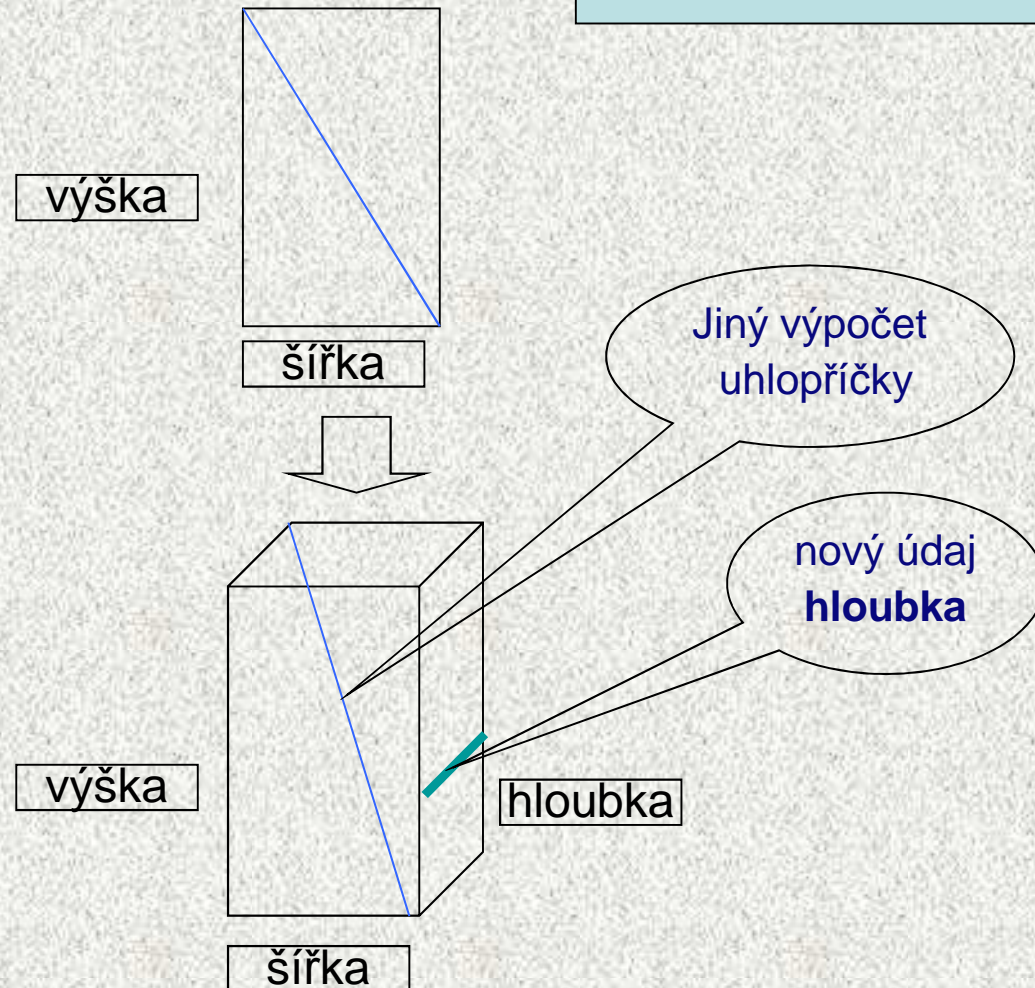
Dědičnost – Kvádr je rozšířený obdélník?



Kvadr extends Obdelnik



Kvadr 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 konstrukturu 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 konstrukturu, 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?

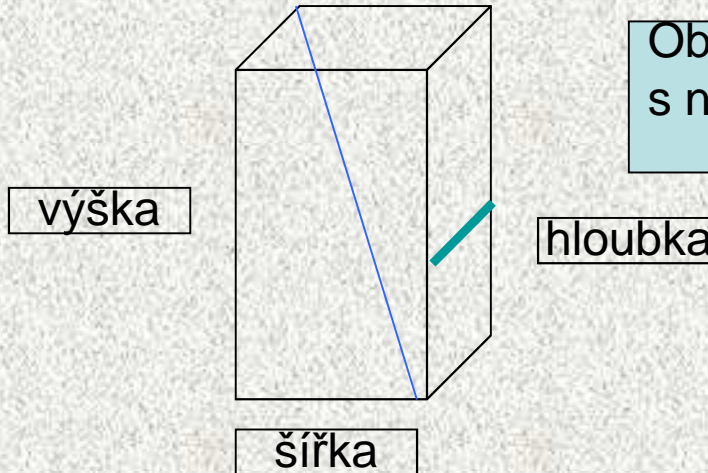
- Jaká je plocha kvádru?
- Jaký je obvod kvádru?
-

Opravdu
?

Je to správná
logika
?

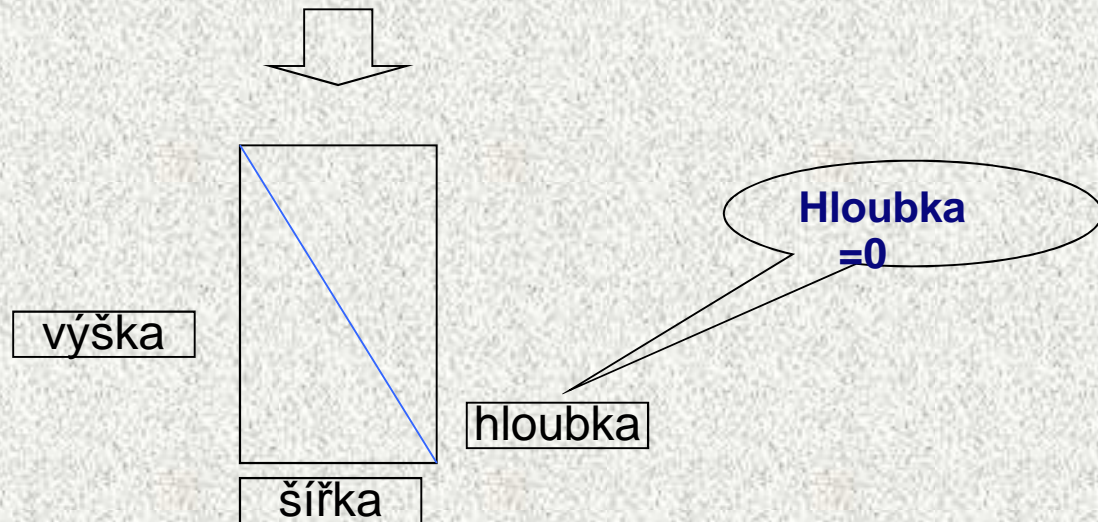
Dědičnost – Obdélník je speciální kvádr?

Kvadr
sirka;
vyska;
hloubka
delkaUhlopričky()
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);
}
}
```

A0B36PR1 - 08

Dědičnost – Obdélník je speciální kvádr?

```
class Obdelnik2 extends Kvadr2{  
  
public Obdelnik2(int sirka, int vyska) {  
super(sirka, vyska, 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??**
 - Co když bude mít obdélník metodu „**prodluž stranu a**“ nebo/i metodu „**prodluž stranu b**“?

Je obdélník potomek kvádru či naopak?

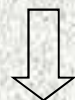
1. Kvádr je potomek obdélníka
 - Logické přidání rozměru, ale metody platné pro obdélník nefungují pro kvádr („vepište elipsu do obdélníka“, „obsah obdélníka“,..)
 2. Obdélník je potomek kvádru
 - Logicky správná úvaha o „specializaci“ („vše co funguje pro kvádr, funguje i kvádr s nulovou hloubkou“), ale neefektivní implementace (každý obdélník je reprezentován 3 rozměry)
- **Odpověď 2 je správná?**
 - A co objem obdélníka?
 - Podobná situace: Komplexní číslo vs. Reálné číslo

re	im
----	----



re	0
----	---

re



re	im
----	----

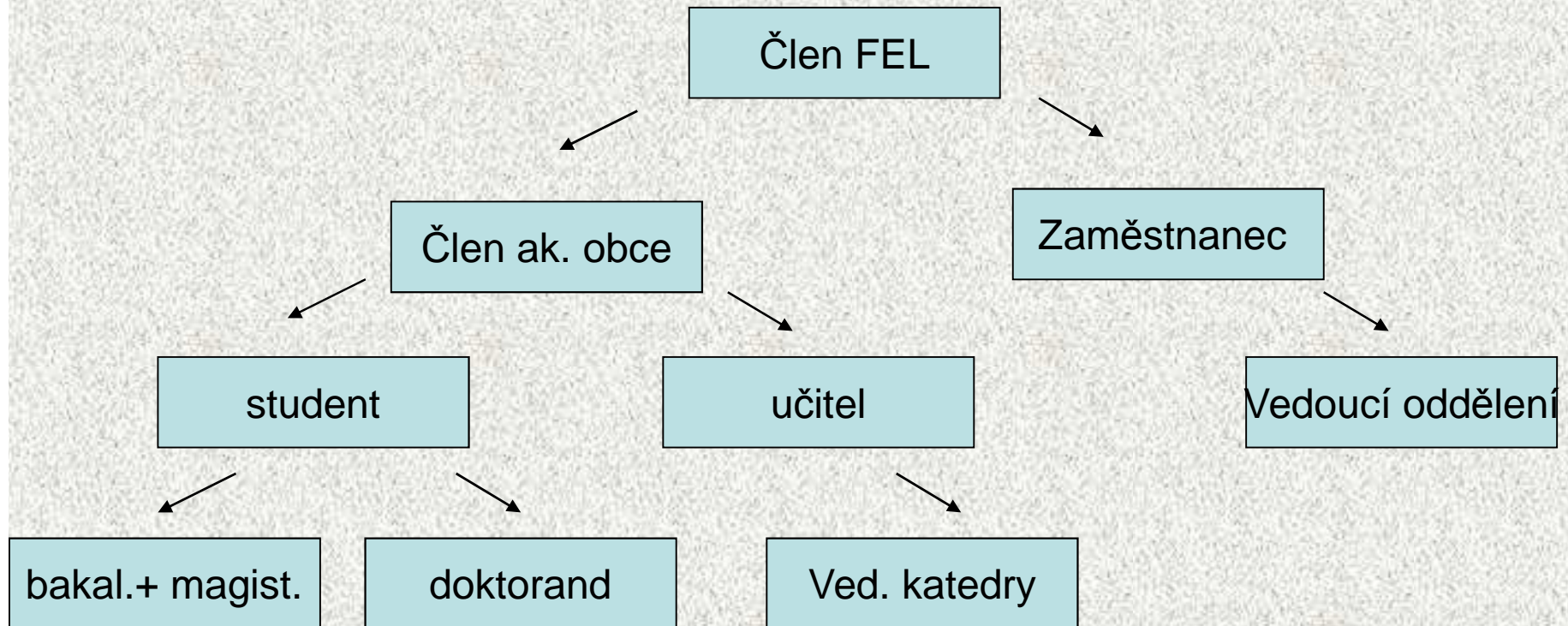
Vše, co platí pro **tátu**,
musí platit pro **syna**

Vše, co platí pro **kompl. č.**,
platí pro **kompl. č.** s im.č = 0,
tedy pro reálné č.

Předek či potomek, vztah „ISA“

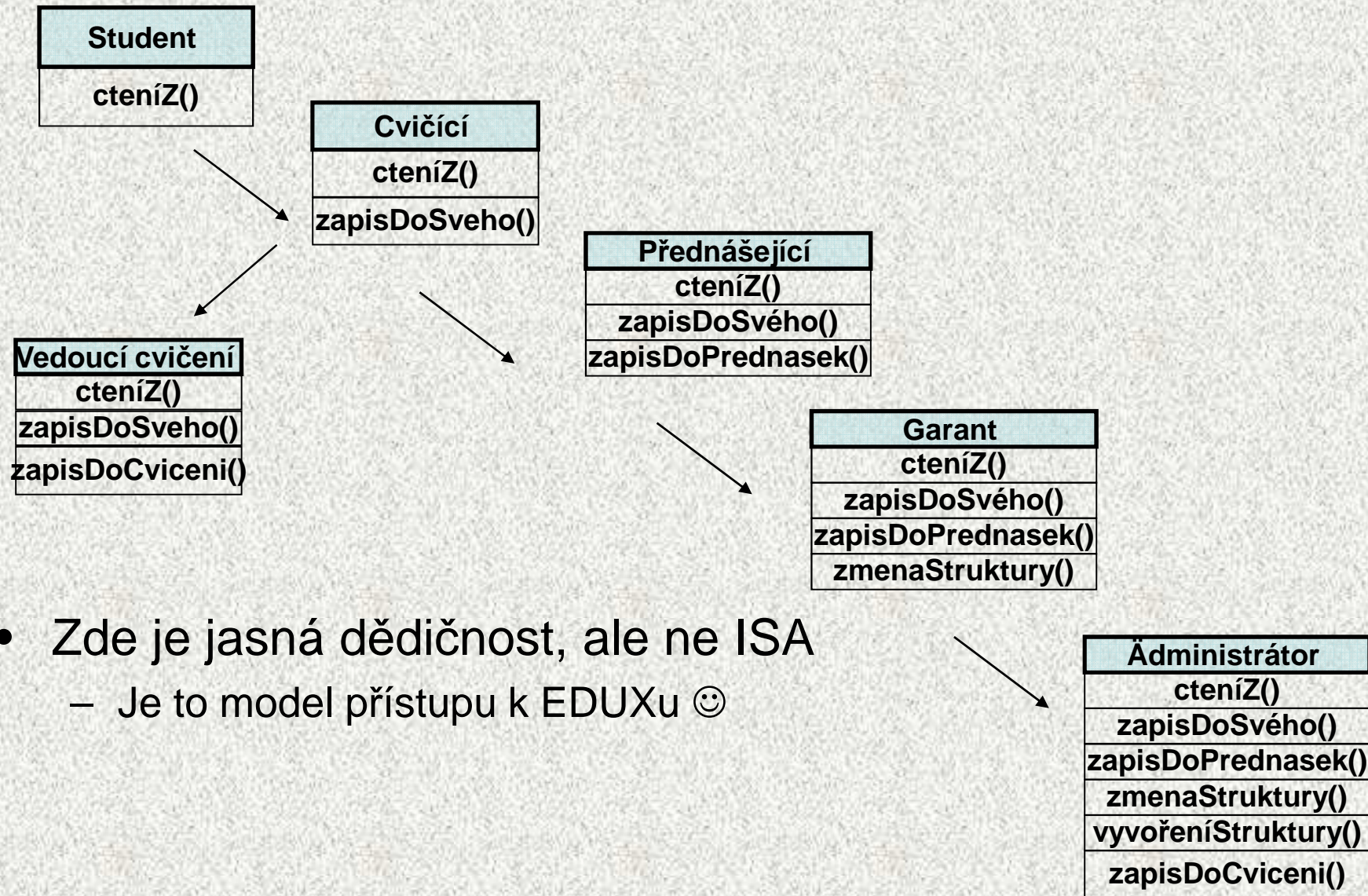
- Je Usecka potomkem Bodu?
 - + Usecka rozšiřuje Bod o jeden jeho konec
 - Usecka nevyužije ani jednu metodu Bodu
 - **ISA vztah**: ? Usecka je Bod? – **NE** → Usecka není potomkem Bodu
- Je Obdelnik potomkem Usecky?
 - **ISA vztah** → **NE**
- Je Obdelnik potomkem Ctverce nebo naopak?
 - Obdelnik rozšiřuje Ctverec o další rozměr, ale není Ctverec
 - Ctverec je Obdelnik, který má šířku i výšku stejnou

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 proměnných) 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:

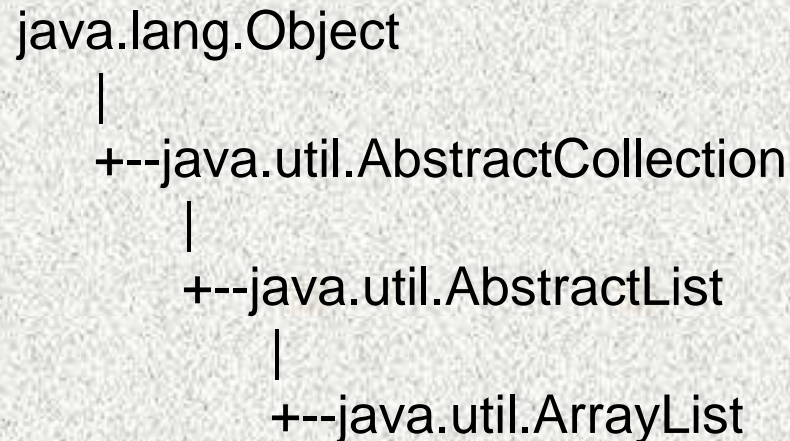
- Východisko polymorfismu
- Jazyk Java zná dědění od jediného předka!
- potřeba zdědit vlastnosti od více předků se řeší rozhraním (**interface**)
– viz další přednášky

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:



- 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*

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  
• ...// další viz další výklad  
...
```

Metody toString (rozšíření)

```
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: prednaska.Obdelnik@1fb8ee3
Druhy: prednaska.Obdelnik@61de33
Stejne? false
Stejne? false

Zastíněné funkce

Prvni: Obdelnik: 5 x 7
Druhy: Obdelnik: 5 x 7
Stejne? false
Stejne? true

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

- Jestliže se překrývá metoda `equals`, pak je nutné překrýt i metodu `hashCode()`, důvody v PR2
- 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";}
```

i= 8 Potomek

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

Dědění - operátor super

```
class BodX {
    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 BodX {
    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 10.63014581273465  
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*
 - hodnotu referenční proměnné typu *Tnad* lze přiřadit referenční proměnné typu *Tpod* pouze 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

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 ciska = new ArrayList();
ciska.add( new Integer(10));
ciska.add( new Integer(20));
System.out.println(ciska.get(0));      // vypíše se 10
System.out.println(ciska.get(1));      // vypíše se 20
Integer prvni = (Integer)ciska.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):

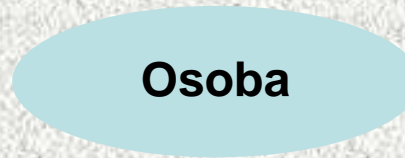
- Jménem
- Adresou
- Datum narození
- Datum nástupu

Datum je charakterizováno těmito atributy(třída Datum):

- Den
- Měsíc
- Rok

Kompozice objektů - příklad

```
class Osoba {  
String adresa, jmeno;  
Datum narozen, nastup;  
...}
```

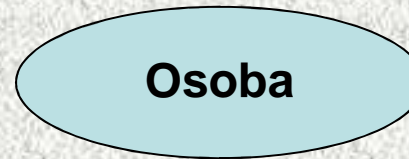


```
class Datum {  
int den, mesic, rok;  
...}
```



Kompozice objektů - příklad

```
class Osoba {  
String adresa, jmeno;  
Datum narozen, nastup;  
...}
```

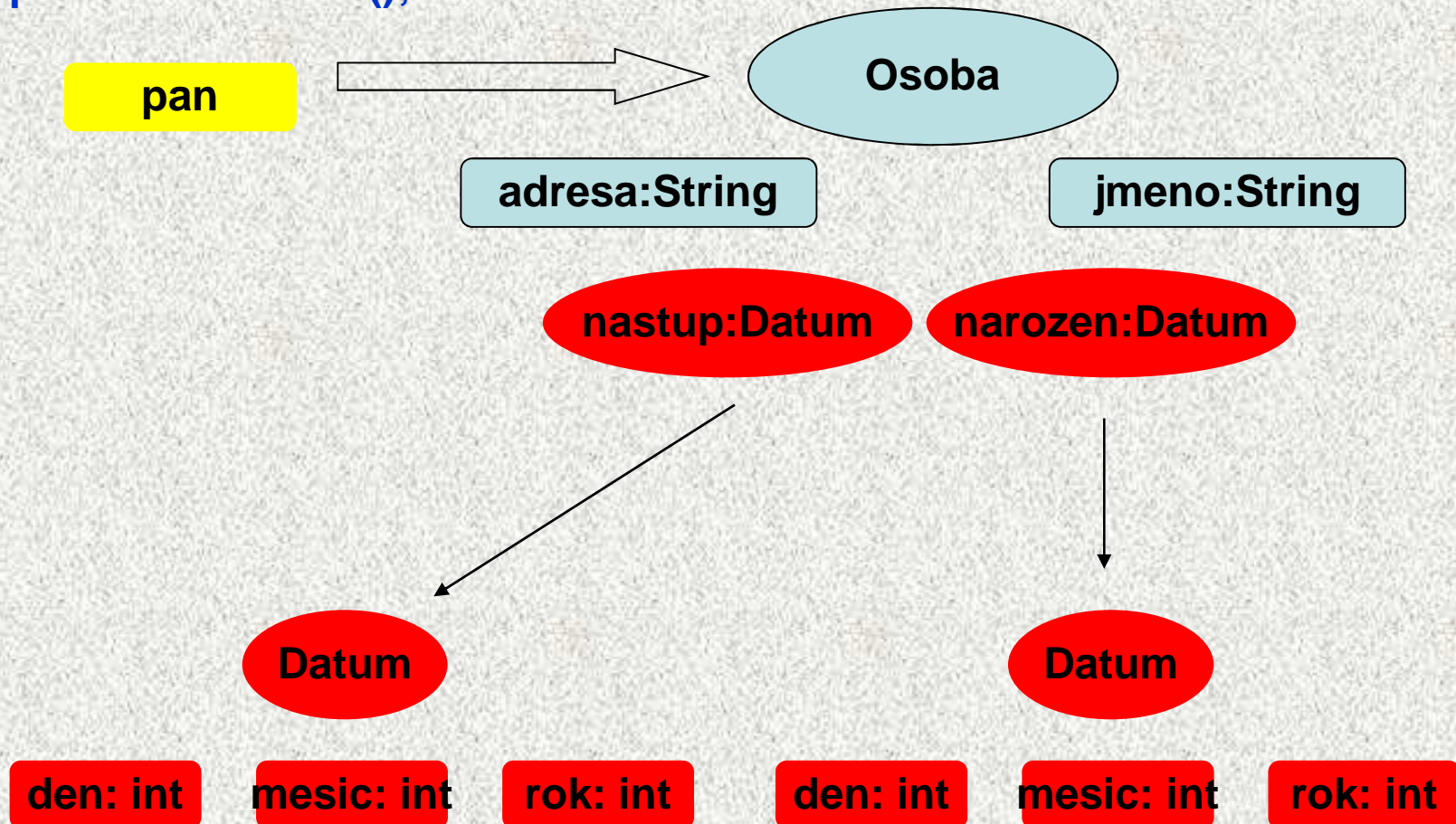


```
class Datum {  
int den, mesic, rok;  
...}
```



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,..){  
    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()
... pan.narozen.getRok() ... pan.nastup.getRok());

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**)

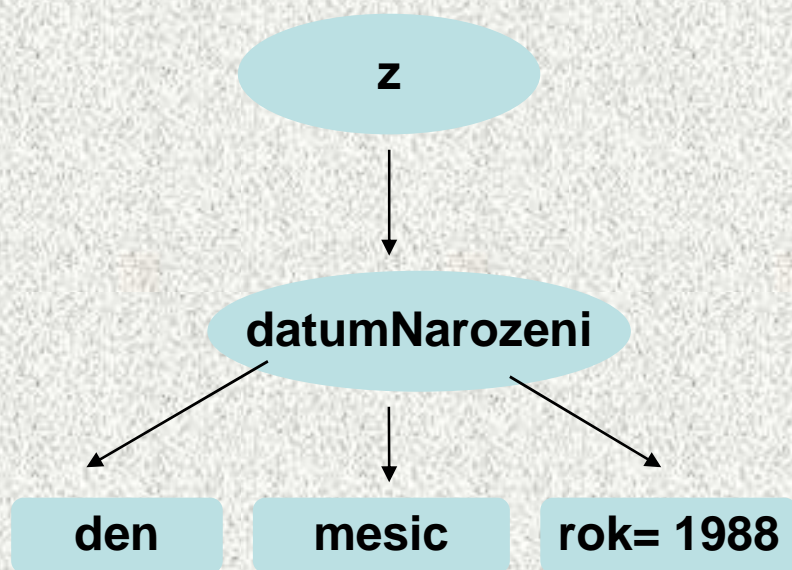
```
class Zamestnanec {  
Datum datumNarozeni = new Datum(); // HAS  
...}
```

```
class Reditel extends Zamestnanec{ // ISA  
Datum datumPovyseni = new Datum // HAS  
}
```

- **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 = 1988;
```

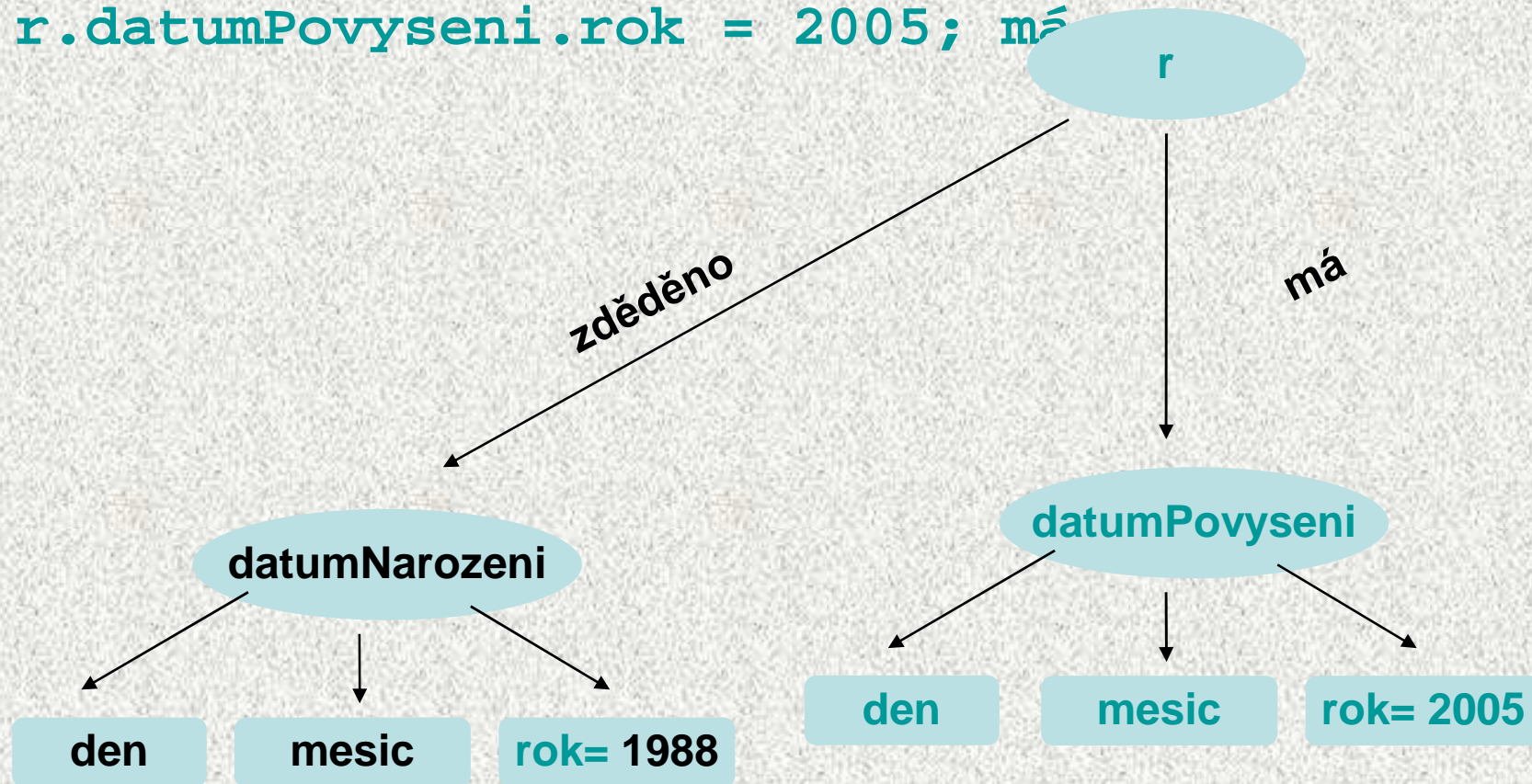


Dědičnost x kompozice - příklad

```
Reditel r = new Reditel();
```

```
r.datumNarozeni.rok = 1978; má dědí
```

```
r.datumPovyseni.rok = 2005; má
```



Dědičnost x kompozice

- Základní vlastnosti dědění objektů:
 - vytváření odvozené třídy (potomka, dceřinou třídu, podtřídu)
 - podtřída se vytváří pomocí `extend` :

```
class Reditel extends Zamestnanec{}
```
 - odvozená třída je specializovanější
 - přidané proměnné (překrytí proměnné)
 - přidané či modifikované metody
 - na rozdíl od kompozice mění vlastnosti objektů
 - nové 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

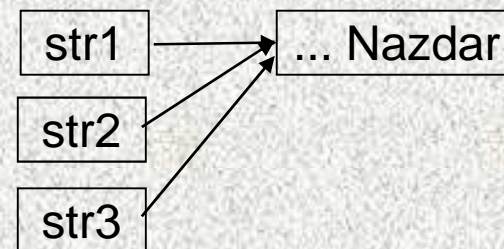
- Objekty knihovny třídy *java.lang.String* jsou řetězky znaků
- Od ostatních tříd se liší třemi specialitami:
 - objekt typu *String* lze vytvořit literálem (posloupnost znaků uzavřená mezi uvozovky, "Pepa", "Další řetězec")
 - hodnotu objektu typu *String* nelze jakkoli změnit
 - operací konkaténace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

- Příklady referenčních 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

Operace s řetězy

- Spojení řetězců (konkatence) +
"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 konkatence řetězců
"abc" + 5 výsledek je "abc5"
"a" + 1 + 2 výsledek je "a12"
"a" + 1 + (-2) výsledek je "a1-2"
- Porovnávání řetězců
 - relační operátory == a != porovnávají reference, nikoliv obsah řetězců
 - pro porovnání řetězců 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
```


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. “kobyła ma mały 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) {
            ...
        }
    }
}
```

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);
}
```



Dědičnost – příklad

```
Obdelnik1 obd = new Obdelnik1(6, 8);  
Kvadr1 kva = new Kvadr1(6, 8, 10);  
System.out.println("Kvadr je rozsirenim obdelnika");  
System.out.println("Uhlopricka obdelnik: „+obd.delkaUhlopricky());  
System.out.println("Uhlopricka kvadr: " + kva.delkaUhlopricky());  
System.out.println("Sirka kvadru je: " + kva.hodnotaSirky());  
System.out.println("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.out.println("Uhlopricka obdelnik: „+obd2.delkaUhlopricky());  
System.out.println("Uhlopricka kvadr: " + kva2.delkaUhlopricky());  
System.out.println("Sirka kvadru je: " + kva2.hodnotaSirky());  
System.out.println("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
```