

Polymorfismus, dědičnost

Karel Richta a kol.

Katedra počítačů
Fakulta elektrotechnická
České vysoké učení technické v Praze

© Karel Richta, 2016

Objektové modelování, A7B36OMO
10/2016, Dědičnost

<https://cw.fel.cvut.cz/wiki/courses/a7b36omo/start>



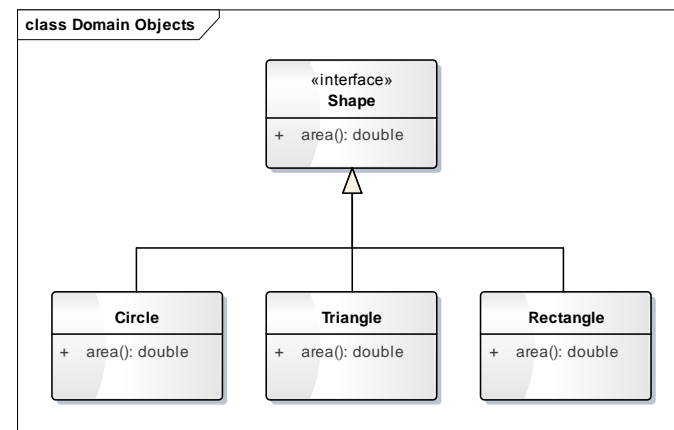
Co je polymorfismus?

- Obecně se polymorfismem rozumí schopnost vyskytovat se v mnoha formách.
- Polymorfismus v jazyku Java:
 - Schopnost reference chovat se dle skutečně odkazovaného objektu.
 - Výběr metody, která se volá, probíhá za běhu programu podle toho na který objekt reference odkazuje – přesun zodpovědnosti za provedení metody na objekt.
- V jazyku Java máme dostupný jak podtypový polymorfismus, tak parametrický.
 - Podtypový polymorfismus je realizován pomocí rozhraní a dědičnosti tříd.
 - Parametrický polymorfismus je realizován pomocí generik.
- Podtypový polymorfismus vychází z principu **subsumpce**.

Podtypový polymorfismus

- Je to způsob jak skrýt více implementací za jedno rozhraní.
- Umožňuje aby stejná zpráva byla obsloužena různě v závislosti na konkrétním objektu.
- **Příklad:** Mějme abstraktní třídu geometrický obrys – **Shape**.
- Polymorfismus umožňuje programátorovi definovat různé metody pro výpočet plochy - **area** ve zděděných třídách **Circle**, **Rectangle**, **Triangle**,

Zavolání metody **area** na referenci typu **Shape** vrátí správný výsledek s ohledem na skutečně odkazovaný objekt.



Princip subsumce

- Jestliže objekt x je typu **A** ($x:A$) a platí, že typ **B** je podtypem (specializací) typu **A**, pak je x také objekt typu **B** ($x:B$).
- Jestliže jsou třídy **Circle**, **Rectangle**, **Triangle** specializací typu **Shape**, pak libovolný objekt typu **Shape** je zahrnut v objektech typu **Circle**, **Rectangle**, **Triangle**.
- Jinými slovy: na místo, kde je očekávána instance nějaké třídy, je možné dosadit instanci jakékoli její podtřídy. Odkazovaný objekt se chová podle toho, jaké třídy je instancí. Pokud několik objektů poskytuje stejné rozhraní, pracuje se s nimi stejným způsobem, ale jejich konkrétní chování se liší podle implementace. Protože je tedy náš objekt instancí podtřídy očekávané nadtřídy, jeho rozhraní je nadmnožinou rozhraní nadtřídy.

Princip subsumce (pokr.)

- Tudíž se s tímto objektem bude pracovat stejně jako s instancí jeho nadtřídy, nicméně bude dostupná pouze ta funkcionality, jejíž rozhraní se shoduje s rozhraním nadtřídy.
- Takže pokud v podtřídě definujeme nějaké specifické metody navíc, nebude možné je použít, pokud se její instance předá namísto instance nadtřídy.

Příklad: Zvířata, kočky a psi

<https://is.muni.cz/www/256036/c.txt>

```
class Animal {
    talk() : string; // obecné zvíře nemá, jak se ozvat, tudíž metoda bude abstraktní
}
class Cat {
    talk() : string {
        return "Meow"; // kočka překrývá metodu talk() : string a vrací mňoukání
    }
}
class Dog {
    ownerName : string; // pes má navíc atribut jméno majitele
    talk() : string { return "Woof"; }
    getOwnerName() : string {
        return ownerName; // pes navíc definuje metodu, která vrátí jméno majitele
    }
}
class Test {
    getAnimalToTalk(Animal animal) : string { return animal.talk(); }
}
```

Příklad: Zvířata, kočky a psi (pokr.)

<https://is.muni.cz/www/256036/c.txt>

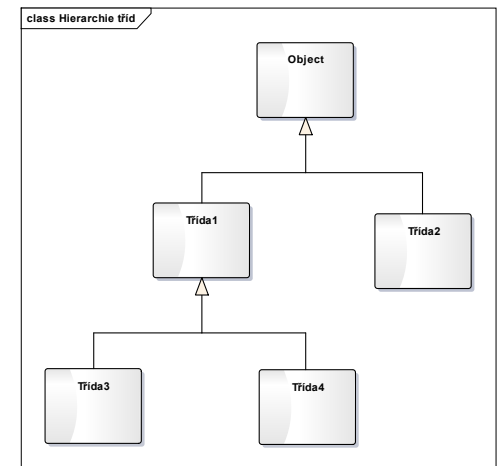
- Metoda **getAnimalToTalk** má parametr typu **Animal**.
- Pokud zavoláme metodu **getAnimalToTalk** s instancí třídy **Cat** jako parametrem (což můžeme díky principu subsumce), dostaneme „Meow“, s instancí třídy **Dog** dostaneme „Woof“.
- Nicméně pokud bychom chtěli v metodě **getAnimalToTalk** navíc zjistit jméno majitele psa, pak bychom ji museli definovat přímo s parametrem třídy **Dog**, protože metoda **getOwnerName** už není součástí rozhraní třídy **Animal**.

Generalizace a specializace

- **Generalizace** je forma asociace, kde třída sdílí strukturu anebo chování jiných/jiné tříd/třídy.
- Definuje hierarchii.
 - Jednoduchá dědičnost
 - Vícenásobná dědičnost
- Je to vztah typu **kind of**.
- **Specializace** je opačná asociace, kdy třída přejímá strukturu a chování jiné třídy a určitým disciplinovaným způsobem ji může měnit – může přidávat nové vlastnosti (ale nemůže žádné ubírat), může přidávat nové metody, příp. modifikovat metody své nadtřídy.
- Je to vztah typu **is a**.

Dědičnost

- Mechanismus jak modelovat generalizaci a specializaci.
- Více specifické třídy (podtřídy) přejímají chování a strukturu obecnějších tříd (nadtríd).
- Třída dědí atributy, operace a vazby.
- V jazyku Java jsou všechny třídy zděděné z třídy **Object**.
- V jazyku Java je k dispozici jednoduchá dědičnost.
- Hierarchie tříd se znázorňuje obvykle stromem.



Dědičnost (pokr.)

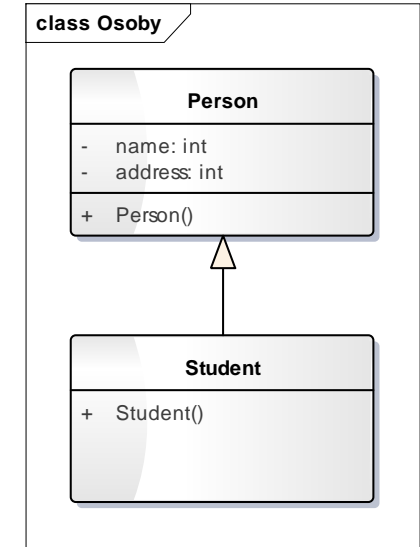
- Výhoda dědičnosti v OOP – znovupoužitelnost.
- Chování (metody) definované v nadtřídě je dostupné ve všech podtřídách.
- Není nutné společnou funkčnost metod definovat znovu.
- Podtřídy mohou implementaci metod měnit a přidávat metody nové.
- V jazyku Java používáme pro specifikaci dědičnosti klíčové slovo ***extends*** u třídy, která dědí.

```
class Potomek extends Rodic { . . . }
```

Příklad

```
public class Person {
    protected String name;
    protected String address;
    /** Default constructor */
    public Person() {
        System.out.println(
            "Inside Person:Constructor");
        name = ""; address = "";
    }
}

public class Student extends Person {
    public Student() {
        System.out.println("Inside Student:Constructor");
    }
}
```



Příklad (pokr.)

- Metoda main:

```
public static void main( String[] args ){  
    Student anna = new Student();  
};
```

- Výstup programu:



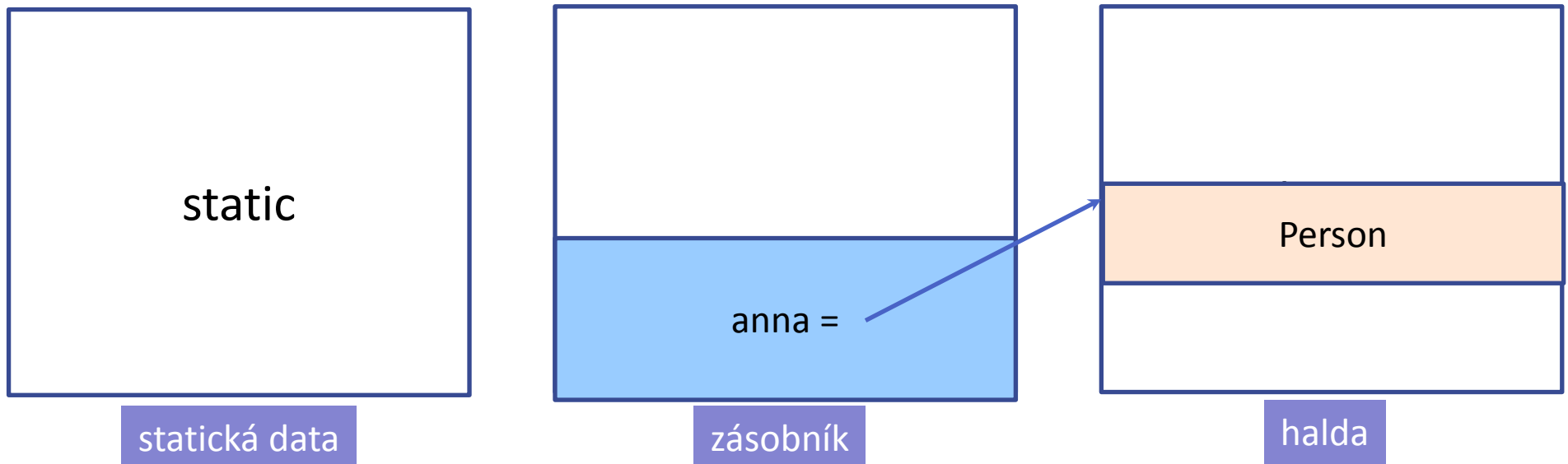
Příklad (pokr.)

- Metoda main:

```
public static void main( String[] args ){  
    Student anna = new Student();  
};
```

- Výstup programu:

Inside Person:Constructor



Příklad (pokr.)

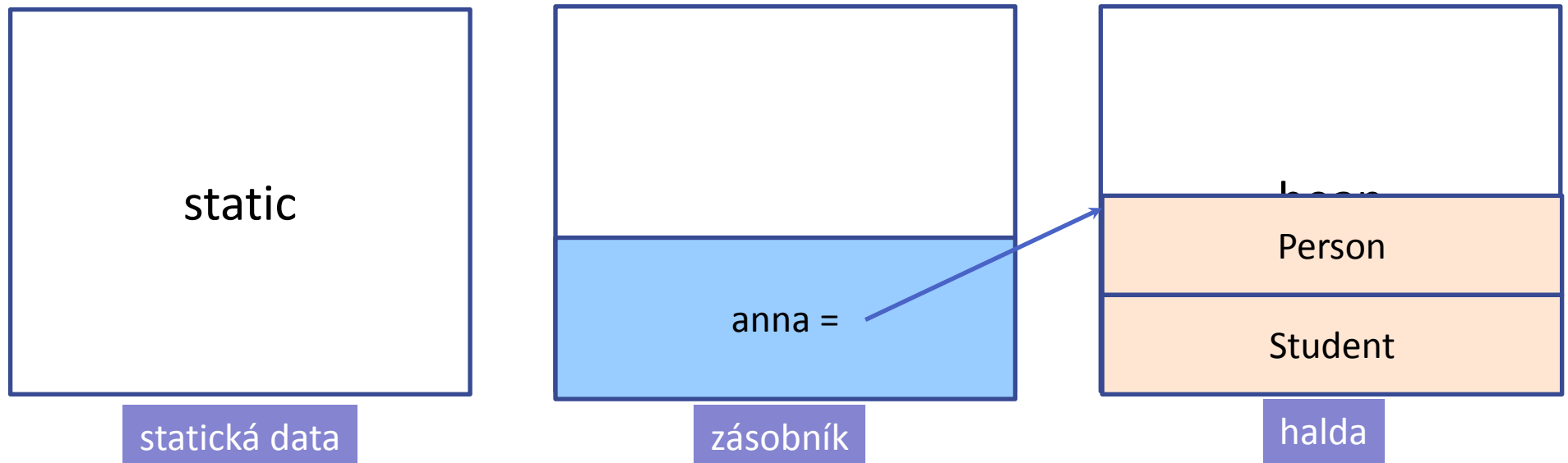
- Metoda main:

```
public static void main( String[] args ){  
    Student anna = new Student();  
};
```

- Výstup programu:

Inside Person:Constructor

Inside Student:Constructor



Příklad (pokr.)

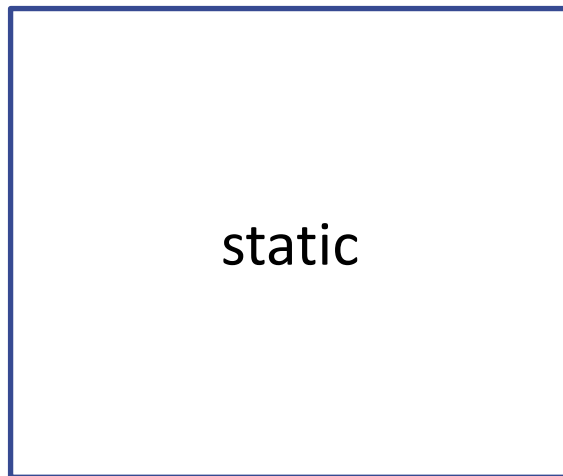
- Metoda main:

```
public static void main( String[] args ){  
    Student anna = new Student();  
};
```

- Výstup programu:

Inside Person:Constructor

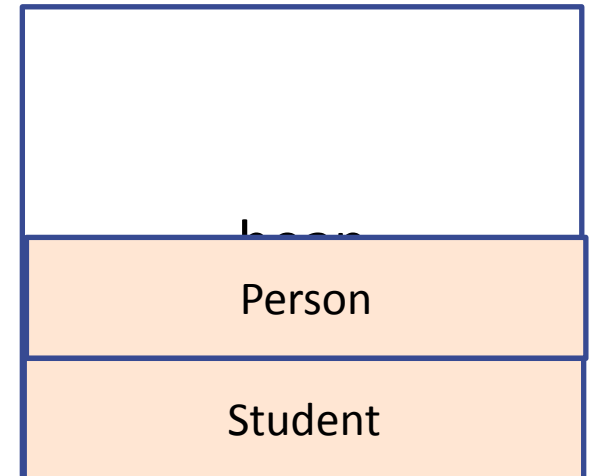
Inside Student:Constructor



statická data



zásobník



halda

Příklad (pokr.)

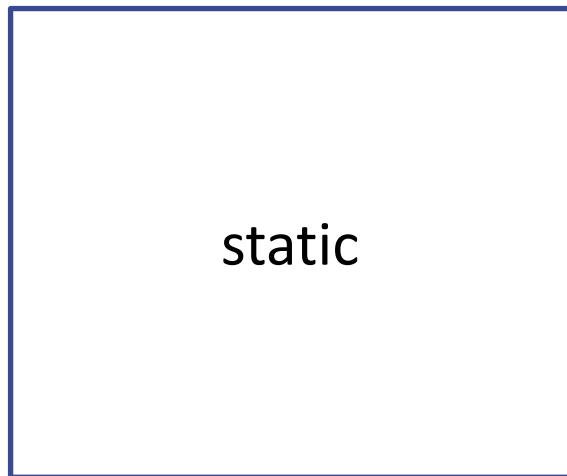
- Metoda main:

```
public static void main( String[] args ){  
    Student anna = new Student();  
};
```

- Výstup programu:

Inside Person:Constructor

Inside Student:Constructor

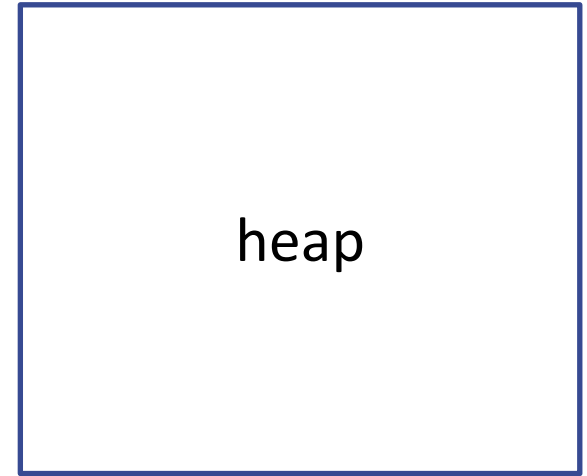


static

statická data



zásobník



heap

halda

Klíčové slovo “super”

- Podtřída může explicitně zavolat konstruktor nejbližší nadtřídě pomocí klíčového slova *super*.

```
public Student(){  
    super( "SomeName", "SomeAddress" );  
    System.out.println("Inside Student:Constructor");  
}
```

- Několik věcí na které je třeba myslet při používání klíčového slova *super* pro volání konstruktoru nadtříd:
 - *super()* musí být jako první příkaz konstruktoru.
 - *super()* může být použito pouze v konstruktoru.
 - Ve stejném konstruktoru nemůžeme použít *this()* a *super()* zároveň.

Klíčové slovo “super” (pokr.)

- Další použití *super* je pokud chceme přistupovat ke členským proměnným nadtřídy.
- Ty ale musí být přístupné.

```
public setStudent() {  
    super.name = "Kdosi";  
    super.address = "Kdesi";  
}
```

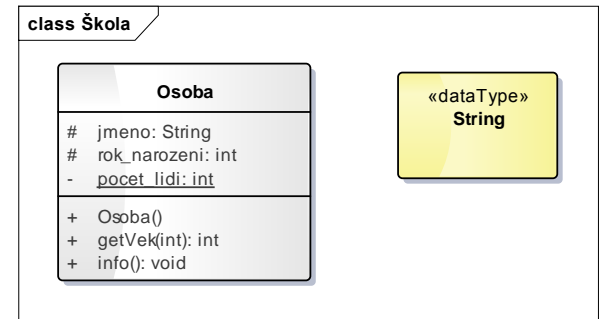
Příklad: Škola

https://moodle.sspbrno.cz/pluginfile.php/10573/mod_resource/content/1/Java-dedicnost-skola.pdf

- Modelujeme-li školu, indikujeme, že se v této skutečnosti vyskytují osoby v různých rolích.
- Některé vlastnosti jsou pro všechny osoby společné, některé speciální typy osob mají vlastnosti, které nejsou pro všechny osoby společné.
- S různými typy osob lze provádět různé akce, podle jejich typu.
- Např. jsou ve škole studenti, kteří jsou zařazeni do různých tříd a mají různé studijní výsledky.
- Nebo jsou zde zaměstnanci, kteří pobírají za svou práci odměnu.
- Speciální typ zaměstnanců jsou učitelé, které škola označuje zkratkou.

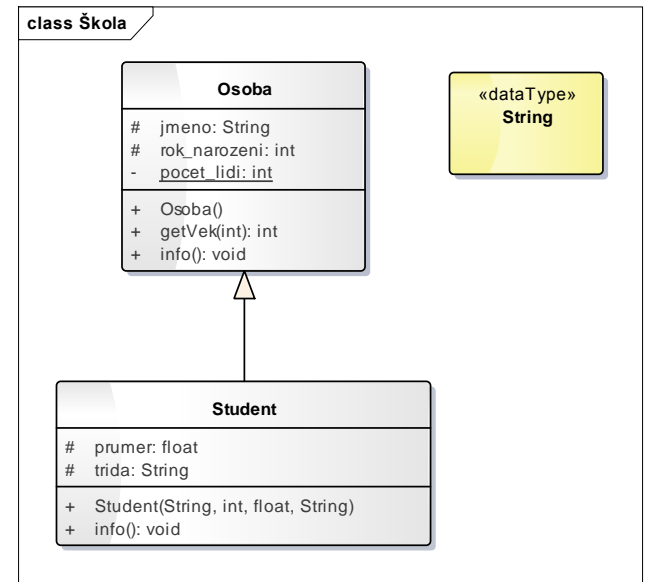
Příklad: Osoby

```
class Osoba {
    protected String jmeno;
    protected int rokNarozeni;
    public static int pocetLidi = 0;
    public Osoba(String jmeno, int rokNarozeni) {
        this.jmeno = jmeno; this.rokNarozeni = rokNarozeni;
        pocetLidi++;
        System.out.println(pocetLidi + ". nový člověk");
    }
    public int getVek(int aktualniRok) {
        return aktualniRok-rokNarozeni;
    }
    public void info() {
        System.out.println(jmeno + " narozen: " + rokNarozeni);
    }
}
```



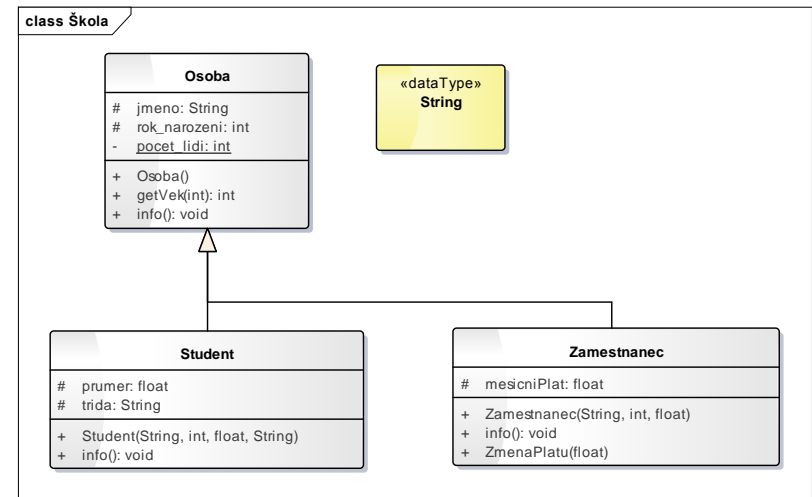
Příklad: Studenti

```
class Student extends Osoba {
    protected float prumer;
    protected String trida;
    public Student(String jmeno, int rokNarozeni,
                   float prumer, String trida) {
        super(jmeno, rokNarozeni);
        this.prumer = prumer;
        this.trida = trida;
        System.out.println("Je to student");
    }
    public void info() {
        super.info();
        System.out.println("třída: " + trida + " průměr známek studenta "
                           + prumer);
    }
}
```



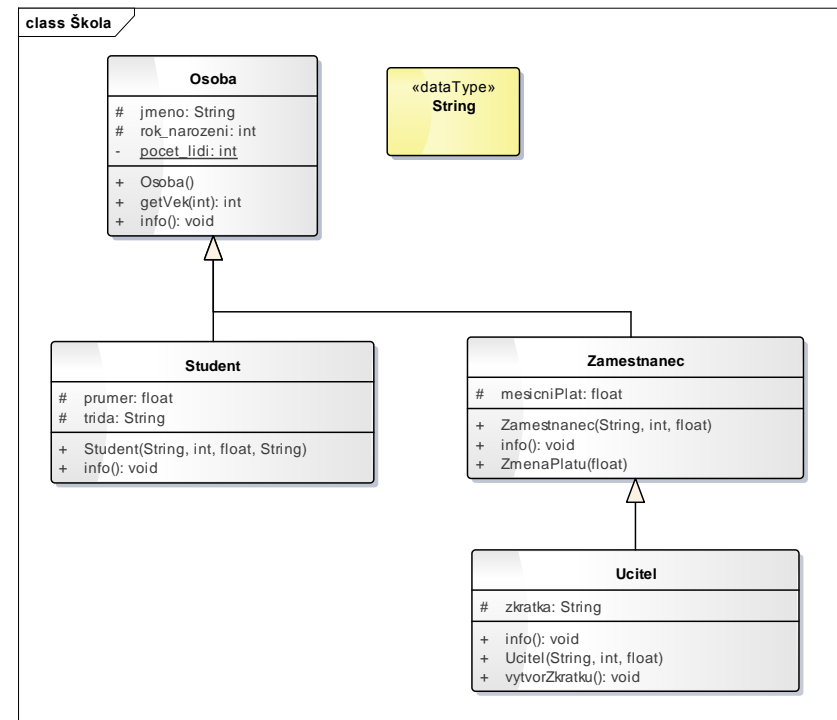
Příklad: Zaměstnanci

```
class Zamestnanec extends Osoba {
    protected float mesicniPlat;
    public Zamestnanec(String jmeno, int rokNarozeni, float mesicniPlat) {
        super(jmeno, rokNarozeni);
        this.mesicniPlat = mesicniPlat;
        System.out.println("Zaměstnanec");
    }
    public void info() {
        super.info();
        System.out.println(
            "Měsíční plat: " + mesicniPlat);
    }
    public void zmenaPlatu(float procenta) {
        mesicniPlat += mesicniPlat*procenta/100;
    }
}
```

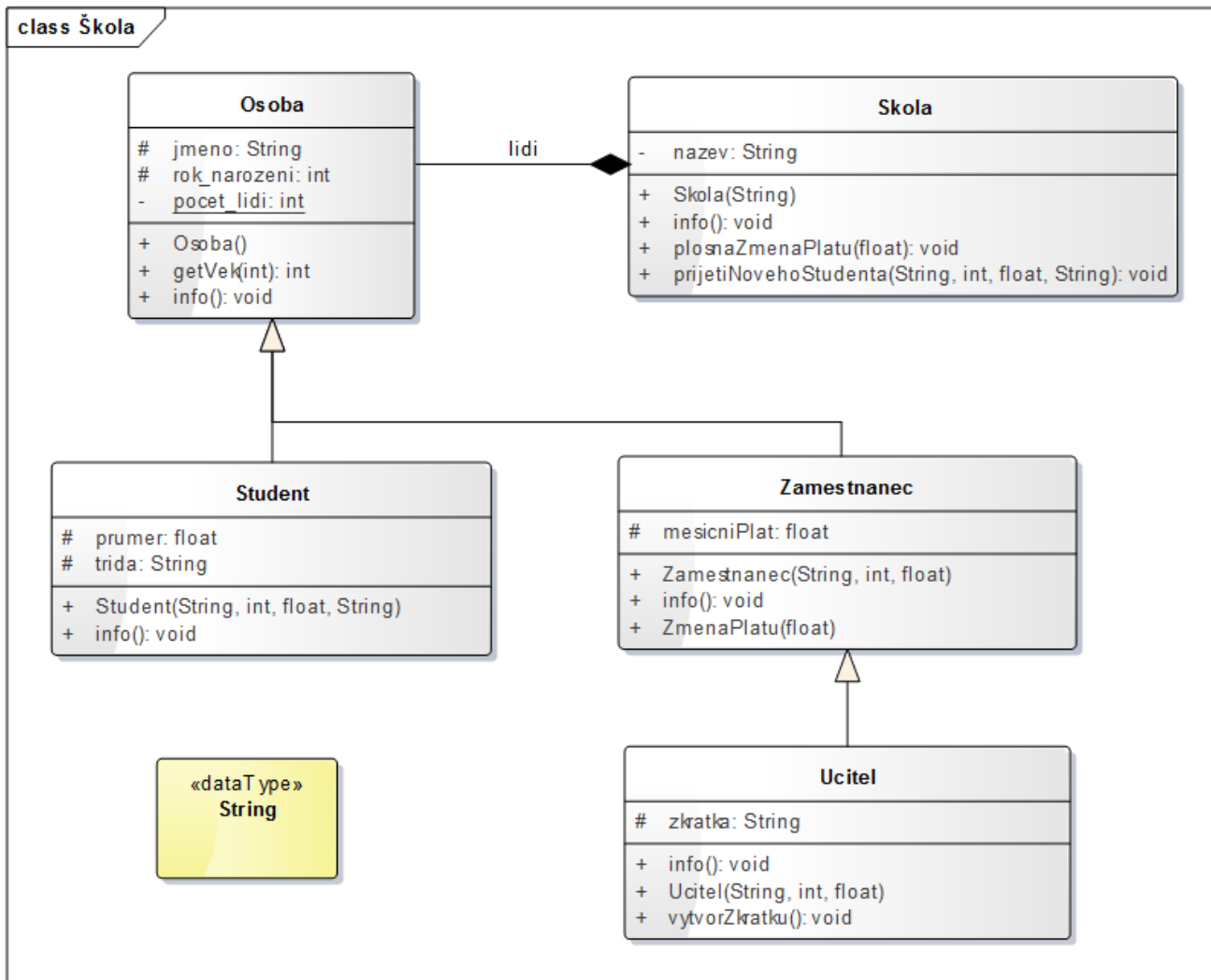


Příklad: Učitelé

```
class Ucitel extends Zamestnanec {
    protected String zkratka;
    public Ucitel(String jmeno, int rokNarozeni, float mesicniPlat) {
        super(jmeno, rokNarozeni, mesicniPlat);
        vytvorZkratku();
    }
    private void vytvorZkratku() {
        zkratka = ""; zkratka += jmeno.charAt(0);
        zkratka += jmeno.charAt(1);
        for (int i=2;i<jmeno.length();i++){
            if(jmeno.charAt(i)==' '){
                zkratka += jmeno.charAt(i+1);
                zkratka += jmeno.charAt(i+2);
                break;
            }
        }
    }
    public void info(){
        super.info(); System.out.println("Zkratka: "+ zkratka);
    }
}
```



Příklad: Škola



Příklad: Škola

```
class Skola {
    private String nazev;
    private Osoba[] lidi = new Osoba[2000];
    public Skola(String nazev) {
        this.nazev = nazev;
    }
    public void prijetiNovehoStudenta(String jmeno, int rokNarozeni, float prumer,
        String trida) {
        lidi[Osoba.pocetLidi] = new Student(jmeno, rokNarozeni, prumer, trida);
    }
    public void prijetiNovehoZamestnance(String jmeno, int rokNarozeni, float plat) {
        lidi[Osoba.pocetLidi] = new Zamestnanec(jmeno, rokNarozeni, plat);
    }
    public void prijetiNovehoUcitele(String jmeno, int rokNarozeni, float plat) {
        lidi[Osoba.pocetLidi] = new Ucitel(jmeno, rokNarozeni, plat);
    }
}
```

Příklad: Škola (pokr.)

```
public void info() {
    for (int i=0; i<Osoba.pocetLidi; i++) {
        lidi[i].info();
    }
}

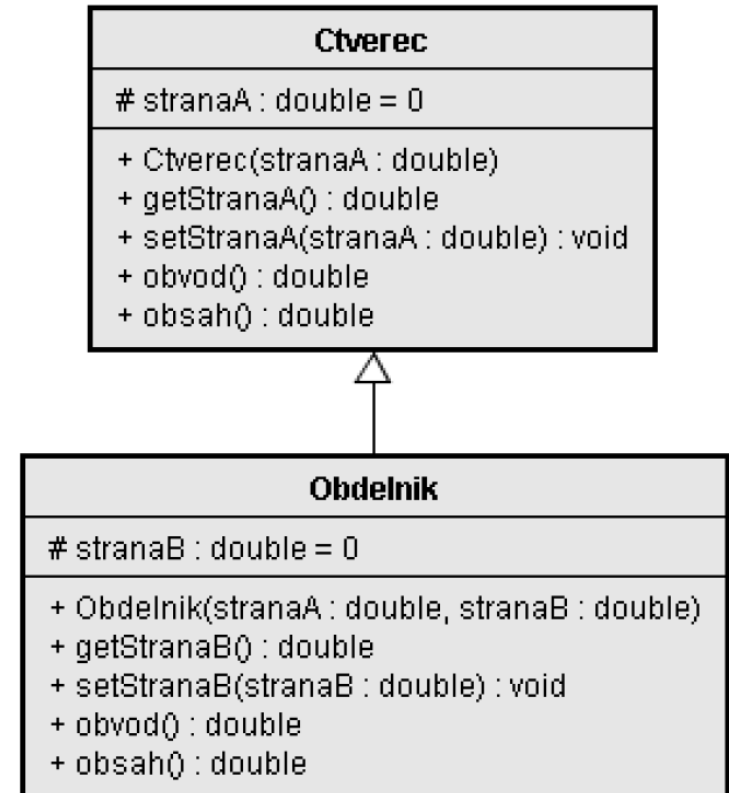
public void plosnaZmenaPlatu(float procenta) {
    for (int i=0; i<Osoba.pocetLidi; i++) {
        String trida = lidi[i].getClass().getName();
        if (trida.contains("Zamestnanec") ||
            trida.contains("Ucitel")) {
            ((Zamestnanec)lidi[i]).zmenaPlatu(procenta);
        }
    }
}
}
```

Příklad použití

```
public static void main(String[] args) {  
    Skola SK = new Skola("FEL");  
    SK.prijetiNovehoZamestnance("Josef Reditel", 1980, 40000);  
    SK.prijetiNovehoUcitele("Karel Ucitel", 1990, 30000);  
    SK.prijetiNovehoUcitele("Jana Ucitelka", 1995, 27000);  
    SK.prijetiNovehoStudenta("Pavel Student", 2010, 2.5F, "1.A");  
    SK.prijetiNovehoStudenta("Dana Studentka", 2011, 1.2F, "1.A");  
    System.out.println("Nase skola: ");  
    SK.info();  
    SK.plosnaZmenaPlatu(10);  
    SK.info();  
}
```

Nevhodné použití dědičnosti

- Dědičnost používáme v případech, kdy odhalíme vztah „býti podtypem“ (potomek je podtypem svého předka – říkáme „is a“).
- Měli bychom si ale také odpovědět na otázku: Je každý potomek také předkem? Pokud ne, není použití dědičnosti vhodné.
- Uvažme vztah mezi čtvercem a obdélníkem.



Problém

- Na první pohled je takto zavedený vztah mezi čtvercem a obdélníkem správně.
- Můžeme správně počítat obvody i obsahy těchto obrazců.
- Zkusme ale zavést novou metodu, která zdvojnásobí plochu čtverce:

```
public void zdvojnásobPlochu(Ctverec c) {  
    C.setStranaA(c.getStranaA() * Math.sqrt(2));  
}
```
- Ta pro čtverce funguje správně, ale pro obdélníky nikoliv.
- Přitom ji lze na obdélník použít, neb je zděděna.
- Problém spočívá v tom, že ne každý obdélník je také čtverec – dědičnost je použita nesprávně.

Příklad

```
class Ctverec {  
    protected double stranaA = 0;  
    public Ctverec(double sA) {  
        stranaA = sA;  
    }  
    public double getStranaA () { return stranaA; }  
    public void setStranaA (double sA) { stranaA = sA; }  
    public double obsah() { return stranaA * stranaA; }  
    public double obvod() { return 4*stranaA; }  
}
```

Příklad (pokr.)

```
class Obdelnik extends Ctverec {
    protected double stranaB = 0;
    public Obdelnik(double sA, double sB) {
        super(sA);
        stranaB = sB;
    }
    public double getStranaB () { return stranaB; }
    public void setStranaB (double sB) { stranaB = sB; }
    public double obsah() { return stranaA * stranaB; }
    public double obvod() { return 2*(stranaA+stranaB); }
}
```

Příklad použití

```
public class Shape {  
    private static Ctverec ct = new Ctverec(10);  
    private static Obdelnik ob = new Obdelnik(7,5);  
    public static void main() {  
        System.out.println("Obvod ctverce: "+ct.obvod());  
        System.out.println("Obvod obdelnika: "+ob.obvod());  
        System.out.println("Obsah ctverce: "+ct.obsah());  
        System.out.println("Obsah obdelnika: "+ob.obvod());  
        System.out.println("Zdvojnásobíme plochu");  
        zdvojnásobPlochu(ct);  
        zdvojnásobPlochu(ob);  
        System.out.println("Obsah ctverce: "+ct.obsah());  
        System.out.println("Obsah obdelnika: "+ob.obvod());  
    }  
}
```


Výstup

Obvod ctverce: 40.0

Obvod obdelnika: 24.0

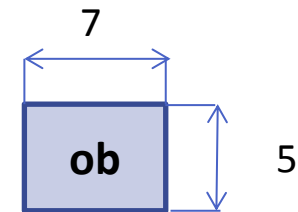
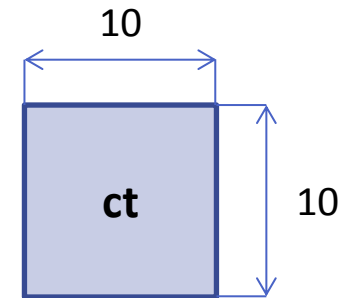
Obsah ctverce: 100.0

Obsah obdelnika: 24.0

Zdvojnásobíme plochu

Obsah ctverce: 200.000000000000003

Obsah obdelnika: 29.79898987322333



Jiný příklad chybné dědičnosti

```
import java.util.*;

public class StackDemo {
    public static void main(String args[]) {
        // creating stack
        Stack st = new Stack();

        // populating stack
        st.push("Java");
        st.push("Source");
        st.push("code");

        // checking stack
        System.out.println("Is stack empty: "+st.empty());
        System.out.println("Druhy element: "+st.elementAt(1));
    }
}
```

Třída Stack je v java.util realizována děděním ze třídy Vector – umí tedy jeho metody.

Pravidla, kdy použít dědičnost

Specializace:

- Jedním z nejčastějších důvodů pro použití dědičnosti je specializace existujících tříd a objektů. Při specializaci získává třída nové datové atributy a chování proti původní třídě. Ukázkou specializace je použití abstraktních tříd.

Překrývání metod a polymorfismus:

- Častým důvodem k dědičnosti je možnost využití překrývání metod a následně polymorfismu – různí potomci mají rozdílně implementovány některé metody. Při volání takové metody programátor nemusí uvažovat o tom, které konkrétní instanci posílá zprávu, neboť každá instance má k sobě přiřazen svůj specifický kód.

Znovupoužití kódu:

- Jednou z prvních motivací pro dědičnost bylo umožnit nové třídě znovu použít kód, který již existoval v jiné třídě. Pokud vede k dědičnosti pouze tento motiv, vznikne hierarchie, kdy věcně nelze přetypovat potomka na předka.

Generické typy a polymorfismus

```
public interface Pair<K, V> {
    public K getKey();
    public V getValue();
}
public class OrderedPair<K, V> implements Pair<K, V> {
    private K key;
    private V value;
    public OrderedPair(K key, V value) {
        this.key = key;
        this.value = value;
    }
    public K getKey() { return key; }
    public V getValue() { return value; }
}
```

Příklad: Zásobník

```
interface StackAPI<T> {
    void push(T x) throws FullStack;
    T pop() throws EmptyStack;
}

class FullStack extends Exception {
    Integer size;
    FullStack(Integer x) { size = x; }
    public String toString() { return "\nStack is full, max size: "+size; }
}

class EmptyStack extends Exception {
    public String toString() {
        return "\nStack is empty";
    }
}
```

Příklad: Zásobník (pokr.)

```
class Stack<T> implements StackAPI<T> {
    public static final int MAX = 10;
    private T items[];
    private int count;
    public void setItems(T x[]) { items = x; }
    public void push(T item) throws FullStack {
        if (count >= MAX) throw new FullStack(count);
        items[count] = item;
        count++;
    }
    public T pop() throws EmptyStack {
        if (count <= 0) throw new EmptyStack();
        count--;
        return items[count];
    }
}
```

Příklad: Zásobník (pokr.)

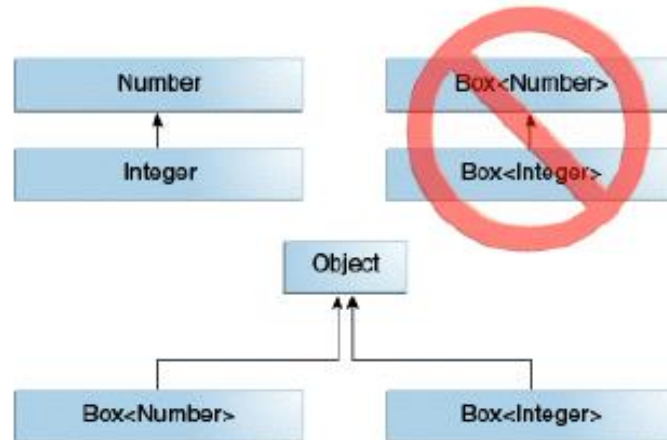
```
public class GenericStack {  
  
    static Integer bodyInteger[] = new Integer[Stack.MAX];  
    static Stack<Integer> stackInteger = new Stack<Integer> ();  
    static Osoba bodyOsoba[] = new Osoba[Stack.MAX];  
    static Stack<Osoba> stackOsoba = new Stack<Osoba> (); //bodyOsoba);  
  
    Integer x = 0;  
    Osoba y = new Osoba("",0);  
  
    public static void main(String[] args) {  
        stackInteger.setItems(bodyInteger);  
        stackOsoba.setItems(bodyOsoba);  
    }  
}
```

Příklad: Zásobník (pokr.)

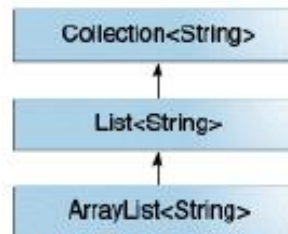
```
try {  
    for (int i = 0; i < 10; i++) {  
        stackInteger.push(i);  
        stackOsoba.push(new Osoba("Jmeno"+i,1999));  
    }  
} catch (FullStack x) { System.out.println(x); }
```

```
try {  
    for (int i = 0; i < 10; i++) {  
        x = stackInteger.pop(); System.out.println(x);  
        y = stackOsoba.pop(); System.out.println(y.jmeno);  
    }  
} catch (EmptyStack q) { System.out.println("Stack is empty"); }  
  
}
```


Generické typy a podtypy



`Box<Integer>` is not a subtype of `Box<Number>` even though `Integer` is a subtype of `Number`.



A sample Collections hierarchy

Porovnávání kompozice a dědičnosti

- Dědičnost obecně znamená "JE" (Is A) - BMW JE Auto
Kompozice obecně znamená "MÁ" (Has a) - Auto MÁ motor
- Dědičnost má v mnoha případech smysl, ale pokud není vztah "X JE Y" zcela evidentní, měl vývojář uvažovat nejprve o kompozici.
- Nevýhody dědičnosti obecně: Strom zděděných tříd je příliš striktní, špatně se mění rozhraní, v kompozici se dá snáze použít odložené spuštění. Na druhou stranu, dědičnost je snazší na použití, třída by měla navenek deklarovat všechny svoje závislosti.

Literatura

- http://fi.muny.cz/data/PB006/PB006_slides2009.pdf
- <https://edux.feld.cvut.cz/courses/X36OBP/media/lectures/02-subtyping.pdf>

The End