Objektově orientované programování

Obsah

- Motivace a kategorie
- Základní informace o třídách, metodách a objektech - Java
- Řízení přístupu k metodám a proměnným
- Předávání a vracení objektů
- Přetěžování
- Dědičnost
- Předefinování metod
- OOP vlastnosti v dalších jazycích
- Implementační principy OOP

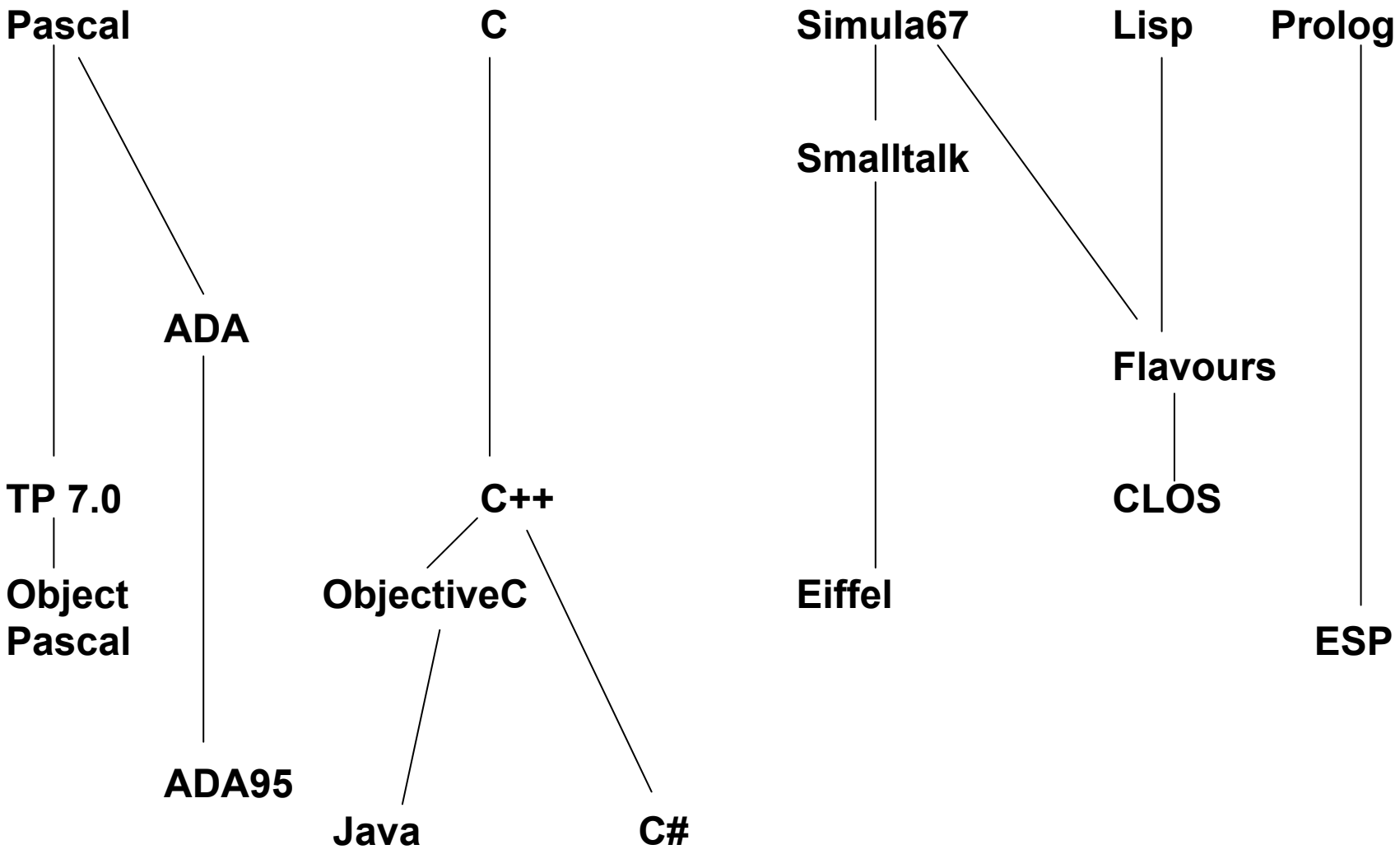
Motivace a kategorie OOP

Softwarová krize
Procedur. orient. programování (procedury)
Datově orient. programování (ADT)
Hlediska při tvorbě software (viz ADT +/-):

- Opakovaná použitelnost komponent (software reuse)
- Modifikovatelnost komponent
- Nezávislost komponent

Prostředky k modifikaci soft. komponent:

- Rozšiřování (fronta →prioritní fronta)
- Omezování
- Předefinování
- Abstrakce
- Polymorfizace (př.print)


Pascal
C
Simula67
Lisp
Prolog
Smalltalk
ADA
Flavours
TP 7.0
C++
CLOS
Object
Pascal
ObjectiveC
Eiffel
ESP
ADA95
Java
C#

Motivace a kategorie OOP

Kategorie OOP jazyků

1. OOP prostředky přidány k existujícímu jazyku
   -C++
   -Ada 95
   -ObjectPascal
   -Scheme
2. Navržen jako OOP jazyk, jeho základní struktura vychází z existujících (imperativních) zvyklostí - Eiffel, nemá přímého předchůdce,
   Java,C# vychází z C++
3. Čistě OOP jazyk
   -Smalltalk

Motivace a kategorie OOP

Proč nevyhovují ADT ?

– navzájem nesouvisí, netvoří hierarchie =
  = obtížně znovupoužitelné

– jejich přizpůsobování je zásah do celého ADT

To je řešitelné pomocí dědičnosti

Základní vlastnosti OOP:

1. <u>Zapouzdřenost</u>
2. <u>Dědičnost</u>
3. <u>Polymorfismus</u>

1.Zapouzdřenost realizuje formou ADT

ADT v OOP je nazýván třídou = šablona pro objekty

Data a procedury jsou společně umístěny ve struktuře zvané objekt = instance třídy:

- modifikovatelnost částí systému
- násobná použitelnost tříd (jako typ i jako předek)
- ukrývání dat
- komunikace objektů = zasílání zpráv

  obsah zprávy:

  -adresát = objekt, kterému se posílá

  -selektor (jméno metody = procedury)
- Vytváření objektů – konstruktor
- Rušení objektů –destruktor

obsah popisu třídy:

-jméno třídy

-deklaraci lokálních a globálních proměnných

-metody použitelné objekty při reagování na zprávy

2.Dědičnost (Třídy mohou dědit data a operace od nadřazených tříd).

- Třída, která dědí je odvozenou třídou = podtřídou
- Rodičovská třída = supertřída
- Dědičnost může ovlivňovat přístup k vnořeným entitám (lze je ukrývat)
- Ve třídě mohou být 2 druhy proměnných – proměnné třídy / proměnné instance třídy
- Ve třídě mohou být 2 druhy metod: metody třídy (zprávy třídě) / metody instance (zprávy objektům)
- Jednoduchá / násobná dědičnost
- Vzniklé závislosti příbuzných tříd komplikují údržbu = nevýhoda
- Pro využitelnost dědičnosti musí třída obsahovat jen jednu logickou entitu

3.Polymorfismus (tatáž zpráva může mít různé významy)

- Třída může metody rodiče nejen zdědit, ale i modifikovat (překrýt = předefinovat)
- Vazba mezi metodou a objektem který ji volá je pak určována až při výpočtu (pozdní, dynamická) = zdržuje
- Abstraktní metody = potomek je musí překrýt
- Finální metody = nelze je překrýt
- Polymorfismus obecně vede k dynamické kontrole typové konzistence = časově náročné

OOP v Javě

Obecné charakteristiky

- Všechna data (kromě primitivních typů) jsou objekty
- Všechny objekty jsou dynamické na haldě a přístupné přes referenční proměnnou
- Jen jednoduchá dědičnost (jeden rodič)
- Všechny metody (kromě final) mají dynamickou vazbu (okamžik určení adresové části příkazu skoku do podprogramu). Důsledky a realizace viz později
- Pro zapouzdření slouží třídy a package (kontejnery tříd)

OOP v Javě –co již víme

-Definice třídy:

```
class JmenoTridy {
    deklarace promennych
    deklarace metod vcetne tela
}
```

-Překladač ukládá každou třídu do separátního souboru

JmenoTridy.class

-Hlavním programem je metoda

```
    public static void main(String args [ ]) { . . . }
```

-Vytvoření instance má tvar:

```
 JmenoTridy jmenoReferencniPromenne = new Konstruktor;
```

-Představuje dva příkazy:

```
 JmenoTridy jmenoReferencniPromenne; //deklarace
  jmenoReferencniPromenne = new Konstruktor; //vytvoreni
                                             //objektu a prirazeni ukazatele
```

OOP v Javě –co již víme

-Není-li konstruktor explicitně definován, vytvoří se implicitní

-Konstruktor se jmenuje jako třída

„ nemá návratový typ

„ používá se i k inicializaci proměnných instance

„ může mít parametry

-Instance objektů jsou na haldě, tu automaticky čistí GC

-Metoda finalize( ) se spustí automaticky, když GC chce zrušit objekt – např. uzavření souborů. Má tvar:

protected void finalize( ) { //operace pred zrusenim }

? Rozdíl mezi destruktorem C++ a Pascalu a finalize

Java nepotřebuje destruktory

OOP v Javě –co již víme

-this je implicitně zavedený ukazatel na objekt, předávaný metodě, kterou objekt volá

např.

```
public class Point {
        public Point(double x, double y) //konstruktor {
                        this.x = x;
                         this.y = y;
        }
                . . .
                private double x;
                private double y;
}
```

Umožňuje stejně pojmenovat lok.prom. nebo parametr jako je jméno proměnné instance

OOP v Javě

Řízení přístupu k metodám a proměnným
-public - je „implicitní“ (pokud nejsou balíky), public elementy lze používat mimo třídu,
-private – jsou přístupny jen metodám definovaným uvnitř jejich třídy
-protected – přístupné i v podtřídách

přístupová práva k proměnným a metodám

Specifikátor	v téže třídě	v pod-třídě	v témže balíku	v podtřídách různých balíků	odkudkoliv
private	+	-	-	-	-
protected	+	+	+	+	-
Neuveden	+	+	+	-	-
public	+	+	+	+	+

Př. 1QDemo Předávání argumentů metodě
-jednoduché datové typy předávány hodnotou
-předávání objektů realizováno odkazem

```
class Queue {  //fronta pro znaky s privátními elementy
   private char q[]; // array pro umístění fronty
   private int putloc, getloc; //indexy vložení,výběru
   Queue(int size) {
      q = new char[size+1]; //alokace paměti pro frontu
       putloc = getloc = 0;
    }
   void put(char ch) {// Put znak do fronty
     if(putloc==q.length-1) {
     System.out.println(“ fronta je plná.");
     return;
     }
     putloc++;
     q[putloc] = ch;
   }
   char get() {// Get znak z fronty.
   if(getloc == putloc) {
     System.out.println(“ Fronta je prázdná.");
     return (char) 0;
   }
   getloc++;
   return q[getloc];
}}
```

```
class QDemo {
  public static void main(String args[]) {
    Queue bigQ = new Queue(100);
    Queue smallQ = new Queue(4);
    char ch;
    int i;

    System.out.println("uziti bigQ k uloz. abecedy");
    // vlozeni do bigQ
    for(i=0; i < 26; i++)
      bigQ.put((char) ('A' + i));

    // vybrani a zobrazeni elementu z bigQ
    System.out.print("Obsah bigQ: ");
    for(i=0; i < 26; i++) {
      ch = bigQ.get();
      if(ch != (char) 0) System.out.print(ch);
    }
    System.out.println("\n");
```

```
System.out.println("uziti smallQ ke gener.preplneni");
 for(i=0; i < 5; i++) {
   System.out.print("Pokus o ulozeni " +
                    (char) ('Z' - i));

   smallQ.put((char) ('Z' - i));

   System.out.println();
 }
 System.out.println();

 // další možné chyby . . .
 System.out.print("Obsah smallQ: ");
 for(i=0; i < 5; i++) {
   ch = smallQ.get();

   if(ch != (char) 0) System.out.print(ch);
}  }  } // konec 1QDemo.java
```

Př 2ErrInfo Typem návratové hodnoty metod může být jakýkoliv typ i typ třída

```
class Err {
  String msg; // chybova zprava
  int dulezitost; // dulezitost chyby

  Err(String m, int s) {
    msg = m;
    dulezitost  = s;
  }
}

class ErrorInfo { // jeji objekty vypisuji chybove zpravy
  String msgs[] = {
    "chyba vystupu",
    "chyba vstupu",
    "plny disk",
    "index mimo meze"
  };
  int stupen[] = { 3, 3, 2, 4 };
```

```
 Err getErrorInfo(int i) { // typem návratové hodnoty je třída
    if(i >=0 & i < msgs.length)
      return new Err(msgs[i], stupen[i]); // vrací objekt
    else
      return new Err("Neplatna chyba", 0);
  }
}

class ErrInfo { // hlavni program
  public static void main(String args[]) {
    ErrorInfo err = new ErrorInfo();
    Err e;
     e = err.getErrorInfo(2); // e přiřazujeme instanci vytvořenou met.getErrorInfo
    System.out.println(e.msg + " dulezitost: " + e.dulezitost);
     e = err.getErrorInfo(19);
    System.out.println(e.msg + " dulezitost: " + e.dulezitost);
  }
}
```

<u>Přetěžování</u> (ad hoc polymorfismus)

<u>Metod</u> -v jedné třídě lze definovat více metod stejného jména. Musí se lišit typem nebo počtem argumentů, návratový typ k rozlišení nestačí.

<u>Konstruktorů</u>

dovoluje konstruovat objekty různými způsoby

Př.3QDemo

```
class Queue {                    //stejna část jako v 1QDemo
  private char q[]; // pole pro ulozeni fronty
  private int putloc, getloc; // indexy pro vloz/vyber
  // 1.konstruktor fronty delky size
  Queue(int size) {
    q = new char[size+1]; // alokace mista pro frontu
    putloc = getloc = 0;
  }                              //konec stejne casti
```

```
// 2.konstr. fronty dle fronty
 Queue(Queue ob) {
   putloc = ob.putloc;
   getloc = ob.getloc;
   q = new char[ob.q.length];
     //kopirovani prvku
   for(int i=getloc+1; i <= putloc; i++)
     q[i] = ob.q[i];
 }

 // 3.konstr.inicializuje frontu polem
 Queue(char a[]) {
   putloc = 0;
   getloc = 0;
   q = new char[a.length+1];
     for(int i = 0; i < a.length; i++) put(a[i]);
 }
```

//následují metody void put a char get stejné jako v př. 1QDemo

```
// Hlavni program
class QDemo {
  public static void main(String args[]) {
    // vytvoreni prazdne fronty pro 10 mist
    Queue q1 = new Queue(10);

    char name[] = {'E', 'v', 'a'};
    // vytvoreni fronty z pole
    Queue q2 = new Queue(name);

    char ch;
    int i;

    // vlozeni znaku do q1
    for(i=0; i < 10; i++)
      q1.put((char) ('A' + i));

    // konstrukce fronty z jine fronty
    Queue q3 = new Queue(q1);
```

```
// Vypis obsahu
   System.out.print("Obsah q1: ");
   for(i=0; i < 10; i++) {
     ch = q1.get();
     System.out.print(ch);
   }

   System.out.println("\n");

   System.out.print("Obsah q2: ");
   for(i=0; i < 3; i++) {
     ch = q2.get();
     System.out.print(ch);
   }

   System.out.println("\n");

   System.out.print("Obsah q3: ");
   for(i=0; i < 10; i++) {
     ch = q3.get();
     System.out.print(ch);
} } }
```

OOP v Javě –Specifikátor static

-Static proměnnou nebo metodu lze použít nezávisle na kterémkoliv objektu. Př. Je main, volané OS

-Volání statické metody vně její třídy má tvar:

JménoTřídyKdeJeDeklarovaná.jménoStatickéMetody

-Příkaz volání nahradí překladač skokem na její začátek

-Statické proměnné jsou v podstatě globální. Existují v jediné kopii, kterou instance sdílejí (nedojde-li k rekurzi) Vně třídy se zpřístupní zápisem:

JménoTřídyKdeJeDeklarovaná.jménoStatickéProměnné

Místo ---„----- lze použít také jméno objektu

OOP v Javě –Specifikátor static

Omezení pro statické metody:

- Mohou volat pouze jiné statické metody
- Nemají definovaný odkaz this
- Mohou zpřístupňovat pouze statická data

Př.

```
class StaticError {
    int delitel = 10;                    //promenna instance
    static int delenec = 1000;           //staticka promenna
    static int deleni( ) {
        return delenec/delitel;          //chyba –
                                         //neprelozi se
    }
}
```

-static{ blok prikazu } je tzv. statický blok

provede se při prvním zavedení třídy, ještě před jejím použitím. Umožňuje inicializace.

OOP v Javě –Vnořené a vnitřní třídy

Vnořené třídy jsou definovány uvnitř jiné (vnější třídy)
Vnitřní třídy jsou vnořené nestatické

-Vnořené třídy jsou použitelné pouze v jejich uzavírající třídě
-Vnořené třídy mají přístup k metodám a proměnným (včetně privátních) definovaným v uzavírající třídě (opak neplatí) Odkazuje se na ně stejně, jako na ostatní nestatické metody vnější třídy.
-Chceme-li ve vnější třídě použít prom. či met. vnitřní třídy, musíme zde vytvořit její instanci a odkazovat způsobem
    jménoInstanceVnitřníTřídy.jménoElementu

-Třídu je možné vnořit do bloku.Ta pak není přístupná kódu vně bloku

Př. 4LCD demonstruje vnoření třídy do metody main

```
class LCD { //binarni zobrazeni
  public static void main(String args[]) {

    class ShowBits {  // vnorena trida
      int numbits;

      ShowBits(int n) {
        numbits = n;
      }

      void show(long val) {
        long mask = 1;

        // posun 1 vlevo na spravne misto
        mask <<= numbits-1;

        int spacer = 0;
        for(; mask != 0; mask >>>= 1) {
          if((val & mask) != 0) System.out.print("1");
          else System.out.print("0");
```

```
        spacer++;
        if((spacer % 8) == 0) {
          System.out.print(" ");
          spacer = 0;
        }
      }
      System.out.println();
    }
  }

  for(byte b = 0; b < 16; b++) {
    ShowBits byteval = new ShowBits(16);  //Vytvoření instance vnitř.třídy

    System.out.print(b + " binarne: ");
    byteval.show(b); //použití metody z vnitřní třídy ve třídě vnější
  }
 }
}
```

OOP v Javě – Dědičnost

Prostředek pro reusing

Superclass nadtřída
Subclass podtřída = (specializovaná verze nadtřídy),

obecný tvar:
class JménoPodtřídy extends JménoNadtřídy { tělo podtřídy }

Dědění umožňuje:

- Ponechat v potomkovi vše to, co z předka považujeme za užitečné
- Přidat do potomka vše co nám v předku chybí
- Změnit v potomkovi vše co potřebujeme aby fungovalo jinak

Dědění je v Javě jednoduché (jednopohlavní)

OOP v Javě – Dědičnost

Zapouzdření zajistí atribut private při deklaraci

private jménoMetody

private jménoProměnné

Podtřída obsahuje všechny zděděné proměnné a metody, nemůže ale přímo zpřístupnit ty z nich, které jsou private. Pro zpřístupnění privátních z nadtřídy musí podtřída použít k tomu určené metody nadtřídy

Proměnná by měla být privátní v případech:

- Mají-li ji používat pouze metody z její třídy
- Má-li nabývat jen určitých hodnot, kontrolovatelných z její třídy

Zásada – K čemu nemůže, to nezkazí

Př. 5Obrazce

```
class DvouDObrazec { //dedeni-dvourozmerne obrazce
  private double width;  // !!privatni = nepřístupné v potomkovi
  private double height; // !!privatni   „          „          „          „
  // Metody pro pristup z potomka
  double getWidth() { return width; }
  double getHeight() { return height; }
  void setWidth(double w) { width = w; }
  void setHeight(double h) { height = h; }
  void showDim() {
    System.out.println("sirka a vyska jsou " + width + " and " + height);
}}
// Podtrida trojuhelniky
class Trojuhelnik extends DvouDObrazec {
  String style;
  double area() {
    return getWidth() * getHeight() / 2;
  }
   void showStyle() {
    System.out.println("Trojuhelnik je " + style);
} }
```

```
class Obrazce {
  public static void main(String args[]) {
    Trojuhelnik t1 = new Trojuhelnik();
    Trojuhelnik t2 = new Trojuhelnik();
    t1.setWidth(20.0); //metoda zdedena z DvouDObrazec
    t1.setHeight(2.0);
    t1.style = "placaty";
    t2.setWidth(5.0);
    t2.setHeight(15.0);
    t2.style = "hubeny";
    System.out.println("informace pro t1: ");
    t1.showStyle();
    t1.showDim();
    System.out.println("plocha = " + t1.area());
    System.out.println();
    System.out.println("informace pro t2: ");
    t2.showStyle();
    t2.showDim();
    System.out.println("plocha =" + t2.area());
  }
}
```

OOP v Javě – Dědičnost

Dědění konstruktorů

- Konstruktor nadtřídy vytváří část objektu patřící nadtřídě
- Konstruktor podtřídy vytváří část objektu patřící podtřídě
- Pokud není konstruktor uveden, uplatní se implicitní
- Definuje-li konstruktor nadtřída i podtřída, musí se při provádění konstr. podtřídy vyvolat konstr. nadtřídy (pro vytvoření své části objektu), nelze ale použít

  new JménoNadtřídy(parametry) proč ???

- Zavádí se příkaz

  super(parametry)

  musí být prvním příkazem v konstruktoru podtřídy
- Metoda super může volat kterýkoliv konstruktor nadtřídy (čím rozliší???)
- Metoda super vždy odkazuje na bezprostřední nadtřídu

Př. 6ObrazceKon

```
class DvouDObrazec { //- dedeni konstruktoru
  private double width;
  private double height;
  DvouDObrazec() { //-1.”implicitni” konstruktor
    width = height = 0.0;
  }
   DvouDObrazec(double w, double h) { //2.-konstr.se 2 parametry
    width = w;
    height = h;
  }
  DvouDObrazec(double x) { //3.-konstr.s 1 parametrem
    width = height = x;
  }
  double getWidth() { return width; }
  double getHeight() { return height; }
  void setWidth(double w) { width = w; }
  void setHeight(double h) { height = h; }
  void showDim() {
    System.out.println("sirka a vyska jsou " +  width + " a " + height);
} }
```

```
class Trojuhelnik extends DvouDObrazec { //-podtrida trojuhelnik
  private String style;
  Trojuhelnik() { //-tvar jako implicitni konstruktor
    super();
    style = "";
  }
  Trojuhelnik(String s, double w, double h) { //-konstr. se 3 param.
    super(w, h); // vola 2.konstructor nadtridy
    style = s;
  }
  Trojuhelnik(double x) {
    super(x); //- vola 3.konstructor nadtridy
    style = "stejne siroky jako vysoky";
  }
  double area() {
    return getWidth() * getHeight() / 2;
  }
  void showStyle() {
    System.out.println("Trojuhelnik je " + style);
}}
```

```
class ObrazceKon {
  public static void main(String args[]) {
    Trojuhelnik t1 = new Trojuhelnik();
    Trojuhelnik t2 = new Trojuhelnik("placaty", 20.0, 2.0);
    Trojuhelnik t3 = new Trojuhelnik(4.0);
    System.out.println("informace pro t1: ");
    t1.showStyle();
    t1.showDim();
    System.out.println("plocha = " + t1.area());
    System.out.println();
    System.out.println("informace pro t2: ");
    t2.showStyle();
    t2.showDim();
    System.out.println("plocha = " + t2.area());
    System.out.println();
    System.out.println("informace pro t3: ");
    t3.showStyle();
    t3.showDim();
    System.out.println("plocha = " + t3.area());
    System.out.println();
}} // konec př. 6ObrazceKon
```

OOP v Javě – Dědičnost

Zpřístupnění proměnných a metod nadtřídy pomocí super

- Řeší situaci, kdy jméno proměnné nebo metody nadtřídy je v podtřídě zakryto lokálním jménem, nebo parametrem
- Plní obdobnou funkci jako this (odkazuje ale na objekt nadtřídy)
- Tvar zápisu:

  super.jménoProměnné,

  super. jménoMetody

Př.

// Pouziti super k zpristupneni zakrytych jmen

```
class A {
  int i;
}
class B extends A {
  int i; // zakryva (predefinuje) i z A
  B(int a, int b) {
    super.i = a; // i z A
    i = b; // i z B
  }
  void show() {
    System.out.println("i v nadtride: " + super.i);
    System.out.println("i v podtride: " + i);
  }
}
class PouzitiSuper {
  public static void main(String args[]) {
    B subOb = new B(1900, 2000);
    subOb.show();
} }
```

OOP v Javě – Dědičnost

Víceúrovňová hierarchie tříd umožňuje dědit od více pokolení předků

Pozor: super umožňuje volat pouze metodu bezprostředního předka.
Nelze použít super.super.jmenoMetody( )

Př 7ObrazceHierar

```
class DvouDObrazec { //-hierarchie trid
  private double width;
  private double height;
. . .
class Trojuhelnik extends DvouDObrazec {
  private String style;
. . .
void showStyle() {
    System.out.println("Trojuhelnik je " + style);
  }
}
```

//stejné jako dříve

```
//přidáme novou třídu

class BarevnyTrojuhelnik extends Trojuhelnik {
  private String barva;

  BarevnyTrojuhelnik(String c, String s,
          double w, double h) {
          super(s, w, h);

          barva = c;
  }

  String getBarva() { return barva; }

  void showBarva() {
          System.out.println("barva je " + barva);
  }
}
```

Třída BarevnyTrojuhelnik bude obsahovat všechny proměnné a metody z Trojuhelnik i z DvouDObrazec

```
class ObrazceHierar {
  public static void main(String args[]) {
    BarevnyTrojuhelnik t1 =
        new BarevnyTrojuhelnik("zluty", "placaty", 20.0, 2.0);
    BarevnyTrojuhelnik t2 =
        new BarevnyTrojuhelnik("zeleny", "vysoky", 2.0, 12.0);
    System.out.println("informace pro t1: ");
    t1.showStyle();
    t1.showDim();
    t1.showBarva();
    System.out.println("Plocha = " + t1.area());
    System.out.println();
    System.out.println("informace pro t2: ");
    t2.showStyle();
    t2.showDim();
    t2.showBarva();
    System.out.println("Plocha = " + t2.area());
  }
}
```

Objekt t1 může používat metodu z třídy Trojúhelník i z třídy DvouDObrazec

OOP v Javě – Dědičnost

při dědění nelze zeslabit přístupová práva k předefinovaným metodám a proměnným

možnosti nastavení práv k metodám a k proměnným v potomkovi

potomek / rodič	private	neuvedeno	protected	public
private	+	+	+	+
neuvedeno	-	+	+	+
protected	-	-	+	+
public	-	-	-	+

neplatí na úrovni tříd - public třída může být zděděna neoznačenou a naopak.
(Public třída musí být v souboru stejného jména.java.)

OOP v Javě – dědičnost a kompozice

kromě dědění lze ze stávajících tříd vytvářet nové třídy kompozicí =
=proměnná může být objektového typu

rozlišení:	kompozice	"má komponentu"
	dědičnost	"je případem"

Př.kompozice

```
class Vyrobce { public String jmeno;  public int ico; . . .}
class Vyrobek { public Vyrobce odKoho; public int cena; . . .}
```

Pořadí volání konstruktorů podtřídy a nadtřídy

- Konstruktory se volají v pořadí shora dolů, tj.konstruktor nadtřídy se volá před konstruktorem podtřídy.
- Metoda super musí být provedena jako první příkaz v konstruktoru podtřídy
- Pokud není konstruktor nadtřídy specifikován, volá se na této pozici implicitní konstruktor nadtřídy

```
//Ukazuje, kdy je volan konstruktor
class A {
  A() {
    System.out.println("konstruovani A.");
  }
}
class B extends A {
  B() {
    System.out.println("konstruovani B.");
  }
}
class C extends B {
  C() {
    System.out.println("konstruovani C.");
  }
}
class PoradiKonstr {
  public static void main(String args[]) {
    C c = new C();
  }
}
```

OOP v Javě – dědičnost a referenční proměnná

Kompatibilita proměnných odkazujících na objekty

Java je jazyk se silným typovým systémem, proto:

- Referenční proměnná jedné třídy nemůže obecně odkazovat na objekt jiné třídy, i kdyby měl stejnou strukturu
- Existuje výjimka ze zásad silného typování - !referenční proměnné nadtřídy může být přiřazena referenční proměnná kterékoliv její podtřídy!.

  Důsledky?

  Co vlastně zpřístupní?

  Zpřístupní jen ty části objektu, které patří nadtřídě

Př. 9NadPodRef

```
class X {  int a;   X(int i) { a = i; } } // nadtrida

class Y extends X {   int b;   Y(int i, int j) {     super(j);     b = i; } } //podtrida

class NadPodRef {
  public static void main(String args[]) {
    X x = new X(100);
    X x2;
    Y y = new Y(15, 25);

    x2 = x; // spravne prirazeno
    System.out.println("x2.a: " + x2.a);

    x2 = y; // take mozno priradit
    System.out.println("x2.a: " + x2.a);

    x2.a = 22; // OK
//    x2.b = 21; // Chybne X nema prvek b
  }
}
```

OOP v Javě – dědičnost a referenční proměnná

Uvedené zásady lze využít i při konstrukci objektů z objektu (kopií objektů)

```
class DvouDObrazec { private double width;   private double height;
    . . .
// Konstrukce objektu z objektu
DvouDObrazec(DvouDObrazec ob) {width =ob.width;height =ob.height;
                                }
    . . .
// Podtřída DvouDObrazec pro trojúhelníky
class Trojuhelnik extends DvouDObrazec {   private String styl;
            . . .
// Konstrukce objektu z objektu ze třídy Trojúhelník
  Trojuhelnik(Trojuhelnik ob) {
    super(ob);//předání objektu třídy Trojúhelník konstruktoru třídy DvouDObrazec
    style = ob.styl;
  }
    . . .
```

OOP v Javě – dědičnost a referenční proměnná

Komentář

- Konstruktor nadtřídy DvouDObrazec očekává jako skutečný parametr objekt typu DvouDObrazec,
- ale konstruktor Trojuhelnik mu předá prostřednictvím super objekt třídy Trojuhelnik.
- Funguje, protože referenční proměnná nadtřídy může odkazovat na objekt podtřídy
- Konstruktor DvouDObrazec() inicializuje pouze ty proměnné a metody objektu podtřídy, které jsou v nadtřídě.

OOP v Javě – dědičnost, přetížení a překrytí

Předefinování metod (překrytí/zastinění/overriding)

- Metoda v podtřídě předefinuje metodu v nadtřídě, pokud má stejné jméno a stejný počet a typ parametrů.
- Předefinovaná metoda překryje metodu z nadtřídy.
- Předefinovanou metodu lze v podtřídě zpřístupnit zápisem super.jmenoMetody(pripadne parametry)
- Má-li metoda podtřídy stejné jméno jako metoda nadtřídy, ale liší se v parametrech, dojde k přetížení metody (overloading)

Př 91

```
class A {
  int i, j;   A(int a, int b) { i = a;  j = b; }
  void show() {  System.out.println("i a j: " + i + " " + j);
} }
class B extends A {
  int k;  B(int a, int b, int c) { super(a, b);  k = c;
  }
  void show() { // predefinovani show() z A
    System.out.println("k: " + k);   super.show();
  }
  void show(String s) { // pretizeni show() z A
    System.out.println("s: " + s);
} }

class Override {
  public static void main(String args[]) {
    B subOb = new B(1, 2, 3);
    subOb.show(); // volani show() v B
    subOb.show("nic"); // volani show(String s) v B
} }
```

OOP v Javě – dědičnost a polymorfismus

Předefinování metod je prostředkem *Objektového polymorfismu*

Dynamická identifikace metody (dynamic dispatching)

- **Schopnost rozpoznat verzi volané (předefinované) metody až při výpočtu**
- **Obsahuje-li nadtřída metodu předefinovanou v podtřídě, pak se při odkazech na různé typy objektů (prostřednictvím referenční proměnné nadtřídy), budou provádět různé verze metod.**
- **Rozhodne se na základě typu objektu, na který referenční proměnná odkazuje při volání metody**
- **Samotný typ referenční proměnné není pro identifikaci metody rozhodující**

Př 93Dynam

```
class Super {   void who() {     System.out.println("volani v Super");   }
}
class Sub1 extends Super {
    void who() {  System.out.println("volani v Sub1"); }
}
class Sub2 extends Super {
    void who() { System.out.println("volani v Sub2");  }
}
class Dynam {
  public static void main(String args[]) {
    Super superOb = new Super();
    Sub1 subOb1 = new Sub1();
    Sub2 subOb2 = new Sub2();
    Super supRef;
    supRef = superOb;
    supRef.who();
    supRef = subOb1;
    supRef.who();
    supRef = subOb2;
    supRef.who();
} }
```

OOP v Javě – dědičnost a abstraktní třídy

Abstraktní třídy

- Definují zobecněné vlastnosti (ve formě abstraktních metod), které budou moci podtřídy sdílet
- Abstraktní metody nemají tělo, tzn. Nejsou v abstraktní třídě implementovány
- Podtřídy musí konkretizovat tyto metody (implementovat je předefinováním). Ostatní metody nadtřídy mohou zdědit, předefinovat nebo přetížit.
- Používají se tehdy, když nadtřída není schopna vytvořit smysluplnou implementaci metody a určí jen šablonu
- Abstraktní mohou být jen obyčejné metody (ne konstruktory, static, final)
- Třída obsahující alespoň jednu metodu abstraktní, musí být také označena abstract
- Pokus o vytvoření objektu abstraktní třídy způsobí chybu při překladu.

Př. 94Abstraktni

```
/* Trida Bod je public, musi byt v souboru Bod.java  */
public class Bod
{  public Bod (double x, double y)
   {  this.x = x;
      this.y = y;
   }
   // ...
   private double x;
   private double y;
}

/*trida Obrazec je public abstract, musi byt predefinovana. */
public abstract class Obrazec
{  public Obrazec (Bod c)
   {  center = c; }
   // ...

   public abstract double plocha();
   private Bod center;
}
```

```
/* Trida Kruh je definovana jako public,  musi byt v souboru Kruh.java */

public class Kruh extends Obrazec
{ public Kruh( Bod c, double r)  { super(c);  radius = r;  }
  // ...
  public double plocha()  { return Math.PI * radius * radius; }
  private double radius;
}

/* Trida Ctverec je public, musi byt v souboru Ctverec.java */

public class Ctverec extends Obrazec
{ public Ctverec (Bod c, double w, double h)
     { super(c); sirka = w;  vyska = h;  }
  // ...
  public double plocha() { return sirka * vyska; }
  private double sirka;
  private double vyska;
}
```

```
/* Trida Uzivatel je public, musi byt v souboru Uzivatel.java*/
public class Uzivatel
{ public static void main(String[] args)
  { Bod x,y;
    Obrazec f;
    Ctverec r;
    Kruh c;
    Obrazec pole[ ] = new Obrazec[3];
    x = new Bod(0,0);
    y = new Bod(1,-1);
    r = new Ctverec(x,1,1);
    c = new Kruh(y,1);
    f = r;
    pole[0] = r;   pole[1]= c;    pole[2] = new Ctverec(y, 5, 2);
    System.out.println(f.plocha());
    f = c;
    System.out.println(f.plocha());
    for (int i=0; i < pole.length; i++) System.out.println(pole[i].plocha());
} }
```

OOP v Javě – dědičnost a specifikace final

Final specifikátor se používá pro takové případy, kdy vzhledem k důležitosti metody/třídy chceme zabránit její modifikaci

- final na začátku deklarace metody znemožňuje její předefinování v podtřídě
- final na začátku deklarace třídy znemožňuje vytvářet její podtřídy. Všechny její metody mají automaticky atribut final
- Označení final u metod třídy nevynucuje označení final pro třídu
- Abstraktní třídu nelze deklarovat jako finální
- Lze ale kombinovat public s final a také public s abstract
- final lze použít i na začátku deklarace proměnné, pak znemožňuje měnit její hodnotu, takové proměnné lze pouze přiřadit počáteční hodnotu. Je to vlastně konstanta

Př.

```
class R {
  final void metoda() {
    System.out.println("je to finalni metoda.");
  }
}
class P extends R {
  void metoda() { // chyba! Nelze predefinovat.
    System.out.println(“ilegalni!");
  }
}
```

Př.

```
final  class R {
    //. . .
}

class P extends R { // chyba, nelze vytvořit potomka
    //. . .
}
```

```
class ErrorMsg {                //Př. 95FinalProm   Navraci String objekt.
  final int OUTERR   = 0;              // Kody chyb
  final int INERR    = 1;
  final int DISKERR  = 2;
  final int INDEXERR = 3;
  String msgs[] =
            { "Output Error", "Input Error", "Disk Full", "Index Out-Of-Bounds" };
   String getErrorMsg(int i) {                    // Navrací zpravu o chybe
   if(i >=0 & i < msgs.length)
      return msgs[i];
   else   return "Invalid Error Code";
} }
class FinalProm {
  public static void main(String args[]) {
    ErrorMsg err = new ErrorMsg();
    System.out.println(err.getErrorMsg(err.INDEXERR));
    System.out.println(err.getErrorMsg(err.OUTERR));
    System.out.println(err.getErrorMsg(err.DISKERR));
} }
```

Třída Object

- je implicitní nadtřídou všech ostatních tříd
- proměnná typu Object může odkazovat proto na objekt kterékoliv jiné třídy
- proměnná typu Object může odkazovat na kterékoliv pole (pole jsou odkazovány jako objekty přes refer. proměnné
- třída Object obsahuje následující metody:

✓ Object clone( ) Vytváří nový objekt, který je stejný jako klonovaný objekt
✓ boolean equals (Object jménoobjektu) Testuje rovnost objektů
✓ void finalize( ) Vyvolá se automaticky před tím než GC odstraní objekt
✓ Class getClass( ) Vrací informace o objektu třídy Class, je finální
✓ int hashCode( ) Vrací číslo přiřazené objektu systémem
✓ void notify( ) Obnoví provádění procesu, který čeká na návrat z metody wait() v this objektu. Je finální
✓ void notifyAll( ) Obnoví provádění všech procesů čekajících na wait v this objektu. Je finální
✓ String toString( ) Vrací řetězec popisující objekt z něhož je volána
✓ void wait( ) Čekání na ukončení jiného precesu
✓ void wait(long milisekundy)
✓ void wait(long milisekundy, int nanosekundy)