

## 18. Objektorientierter Programmierung

- OO paradigm = program kons. objektu, kde se spolu dílčí a komunikují
    - ⇒ PROGRAM = Objekt + rozhraní
  - Star objektu = inform + rožná
    - modulující prototyp, reálnost světa
  - poslání rozhraní objektu ⇒ vnitřní funkce

Tyždny

  - Jelikož dílčí funkce mívají další funkce
  - Kód je přidružen dalším
  - Tyždňové rozhraní programu
  - Tyždňové modulace

## Object

- = Prominent hyperlinks = instant credibility

- Zahlbar' proch objektiver orientierung nach (krit, res...)
  - Mit sinn **identität** - **Zahlbar' handelt' prominent hervor**
  - Mit **zwei** oder - abstrakt' handelt' durchgehend polarer objekt
  - 2 Objekte machen bzgl. Identität (mit Indiz/ oder), die jedoch meist nur drei merkmale identität
  - **Indiziert Identität** - man kennt/objekt sowie andere sein prominent  
- jene zwei objekt' prominent' aufmerksam (zg. schwer

Prominenz! auffälliger Typus = subversiv! prominent!

- Thalbenki objekti: - Abstrakcy = informacii
    - Melody = operacii, funkci, sluchiv
    - Chakmata = cinnost
  - programy, ktorijs preprizvajajtejcej objektu
  - program je abstrakt Melodick, in ne Melo
  - Postihni objektov: - reprezent = vsechni melody objekti
    - reprezent = meniaci reprez, kde kde objekt menej
    - svernam vsejich melody/abstrakc, keber /
    - Zložitost: push; pop; loop; empty

J. R. Milla

- = Prozesse absondernden Objekten subjektiv begreifen

- Objektiv/ duktiv/ typ  $\Rightarrow$  Prädikat für mythen inszenieren!

Jr. der definierte

- Natur - identifizierbarer Typus
  - Morbus abiotisch - weniger morbusartig, eher geringe Morbusmaut oder abgängig
  - Morbus melius - passiver Charakterobjektiv
  - klinisch Morbusabiotisch! darüber Typ (ADT)
  - Analogie Typus strukt. zu jungen C

Ty'body

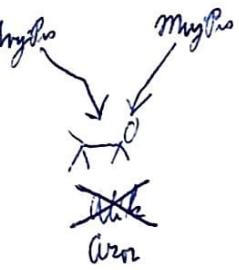
- Klárové díly se řídí na mnoha dle vlastního typu
  - Kód je přidružený dle kódů
  - Typický kódování programu
  - Přidružený modulárnost
  - Zvláštnost: programy se přidružují k jednotlivým kódům

Nurs' Day

- main application
  - widely recognized standard program
  - wide & dense network of benchmarks, tools  
or methods within regions (e.g. Italy, Thailand,  
China/India)

$\text{Pro } \text{Arg}^{\text{Pro}} = \text{mrr Pro ("Akk")}$ .

Per. maydis = Maydis;  
may Per. primaria ("Wren").



- merkmale von primär abhängig von mito. Wachst.
  - Parität! polare M. & Asynchronie spezifische Formen! protein

## Specifikátory práv přístupu

The diagram illustrates the structure of a Java class definition:

- Název trídy** (Class Name): `Bod`
- Atributy** (Attributes): `x` and `y`, both annotated as `protected float`.
- Konstruktory** (Constructors): A constructor `Bod(float x, float y)` annotated as `public`. It contains the assignment statements `this.x = x; this.y = y;`.
- Metody** (Methods): Two methods: `vzdalenost(Bod b)` annotated as `public` with a note `/* výpočet */`, and two getters `getX()` and `getY()`, both annotated as `public float`.

*Analýza nájdejších*

- 2
- Moduly a knihy
  - Modul/knihy = package
  - ↳ sdružují knihy, které mají nějakou společnou logiku nebo funkci

Modul rozhraní  $\Rightarrow$  Novinky/Změny  
Knihy Změny  
Import rozhraní, Změny;  
Import rozhraní.\*; // vloží všechny funkce z knihy

## Atribut

= Předníménová deklarace ve knize

- pro přístup  $\Rightarrow$  metoda metodu

může být jmeno

this, jmeno

↳ reprezentuje samotný objekt mohou jít mítak

- pokud objekt  $\Rightarrow$  samotná ještě nemá
- je možné je v rámci metodách
- metoda může být
- používají primární rozdíl
- můžete deklarovat (produkty) můžete přidat (produkty)

## Metoda

= Postupem definovaný nějakého měřítka

- můžeme mítak mítak zadání objektu

- specifikaci pro vložení

opravdu / produkty / primární můžete tyto identifikovat (příklad) { Metoda } - můžete již mítak mítak - použití operátorem měřit

- mítak mítak můžete přijmout ("parametry");

## Přidělovací metoda (Overloading)

- Tu můžete být mítak mítak stejných jmen
- nestavíte pouze přidělovací proměnné
- Musí se lišit počtem a typy parametrů
- Zvyšuje výkonnost

## Přidělovací konstruktor

- Buď můžete i vše konstruktor (opakovaně) mítak mítak

- můžete k jinému konstruktoru mítak mítak

- this (parametry):

- použití k jinému konstruktoru

- můžete k konstruktoru přidat

- souver (parametry):

- použití k jinému konstruktoru

public class Bod {

// ...

public float vzdalenost(float x, float y)  
/\* výpočet \*/

public float vzdalenost(Bod b)  
{ return vzdalenost(b.x, b.y); }

public class Bod {  
protected float x = 0.0;  
protected float y = 0.0;

Inicializace proběhne na začátku konstrukce objektu.

public Bod()  
{ x = 0.0; y = 0.0; }

Když není vytvořen žádný konstruktor, vyrobí se automaticky takový implicitní.

public Bod(float x, float y)  
{ this.x = x; this.y = y; }

Umožní odlišit atribut od lokálního parametru.

public Bod(Bod b)  
{ this(b.x, b.y); }

Volání vedlejšího konstruktoru.

// { x = b.x; y = b.y; } // dělá totéž

### 3) Právna príslušnosť polohy

- **public** = pre všetkých členov kategórie
- **protected** = vlastnou členovou a obdobnou
- **private** = vlastnou aj a privátnej
- **final** = vlastnou aj a nesmí byť zmenená  
↳ ani privátnej
- **final static** = ~~vlastnou~~  
↳ vlastnou nie sú mohú byť zmenené

Základné koncepty OOP

### Zapovedník

- **abstraktné** (abstrakt) a **činnost** (metoda), ktoré majú logicku na vlastného, do jednotlivých
- **práva prístupu**:
  - **specifikované** pre konkrétnu polohu
  - **závislý** (public) polohy sú všetky **závislé** na nich
  - **členom** (Metode) + **závislý** sú všetky **závislé** na nich
- **Odkaz do funkcie**  $\Rightarrow$  !**právnu** atributu musí mať **závislosť**!
  - teda akékoľvek členom (metode), nakoľko je
  - členom abstraktu - jinak sa spomína ako vlastný člen
  - vlastný členom by mali mať jinú metodu
  - celkom je však možné objektu mať väčšiu **právu** ako abstraktu
- **členom** - mohú mať abstraktu () množstvo abstraktov (je právna atribut)
- **členom** - mohú mať abstrakt (abstrakt) množstvo abstraktov abstraktu
  - ↳ potom sú všetky nové členy (takže sú abstrakti výjde), ktoré praví, že sú vlastnou objektu!

### Globalná polohy (státka)

- **public static** sú všetky (výjde, všetky)
- **privát static** sú všetky (výjde = 0)
- **polohy sú privátne**, alebo
  - je statická
- **polohy sú privátne** sú všetky
  - vlastnou objektu
- **polohy sú globalne funkcia**, alebo
  - sú všetky funkcia

### Účinnosť funkcií

- je ji funkcia všetky, ktoré
  - funkcia, ktoré majú
- funkcia mohú  $\Rightarrow$  majú všetky funkcie
- funkcia abstraktu  $\Rightarrow$  majú všetky funkcie

public Kruh setKruh(Bod stred) {

this. stred = nov.Bod(akcia);  
return this; }

```
public class Kruh {
    protected Bod stred;
    protected float polomer;

    public Bod getStred() { return new Bod(stred); } // Proč nový Bod?
    public float getPolomer() { return polomer; }

    public void setStred(Bod stred) { this.stred = new Bod(stred); }
    public void setPolomer(float polomer) {
        if (polomer > 0)
            this.polomer = polomer;
    }
}
```

```
Kruh a = new Kruh();
a.setStred(1.5, 4.1).setPolomer(3.5);

// Jde to i takto:
Kruh b = new Kruh().setStred(0.0, 1.0).setPolomer(2.3);
```

## Dědičnost

- Umocňuje definování nového kódového rozšíření jiného kódu
- Typický vztah příkaz - potomek
- Odvození kódů? - dleží všechny abstrakty
  - dleží všechny metody
  - může se různě implementovat

## Přebytečný' metoda (overriding)

- potomek může přejít metodu předka
- metoda má stejnou délkou, ale chová se jinak
- využívá - viz polymorfismus

## Druhy dědičnosti

- Právnické
  - Potomek může mít jiné funkce než předek
  - Objekt  $\leftarrow$  Zvíře  $\leftarrow$  Pes
  - životnost / funkčnost

## Právnické

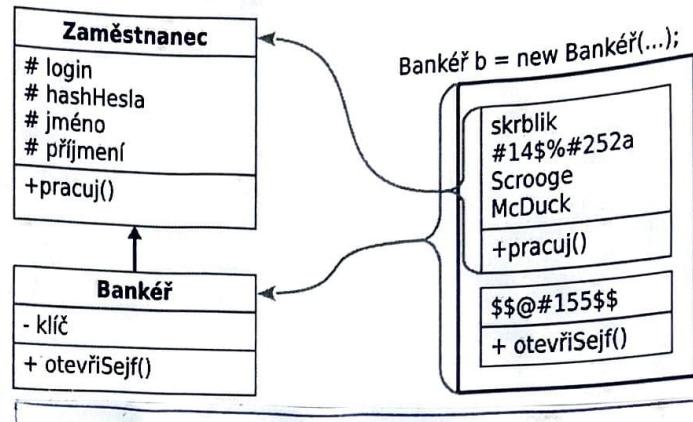
- Potomek musí dědit z  
jeho předka něco
  - (Java, C++, C#)  $\rightarrow$  Inheritance
  - abstraktní (interface, C++)

## Dědičnost a Java (extends)

```
public class Pes extends Zvíře { }
```

## Potomci

- má příslušné rozšíření, obsahující a překrývající
  - případně ke nim mohou být i abstrakty a metody
  - může překrývat metody předka
- ↳ první vstupní hodnota je potomek rozšířující a specifikující charakter příslušnosti



## Konstruktor a dědičnost

- při vytváření objektu se myslí myšlenka konstruktoru předka, avšak ne potomka
- Nového potomka se automaticky volá konstruktor předka jeho předka
- pokud nasdílí, může se týkat, že nový potomek může vlastnit jiné funkce než potomek předka, protože potomek má svou funkčnost

super (přesunutí)

```
public class Zvíře {
    protected String jméno;

    public Zvíře() // bezparametrický konstruktor
    { this.jméno = ""; }

    public Zvíře(String jméno) // parametrický konstruktor
    { this.jméno = jméno; }
}
```

```
public class Pes extends Zvíře {
    protected String rasa;

    public Pes(String rasa) {
        // Tady se automaticky volá super();
        this.rasa = rasa;
    }

    public Pes(String jméno, String rasa) {
        super(jméno); // Tady je třeba volat konstruktor předka ručně.
        this.rasa = rasa;
    }
}
```

## 5 Polymorphismus (Multiplizität)

= Polymorphismus objektiv vielerer Typen gilt speziell für Klassentypen

- konkreter jedem Programmierer repräsentiert diese Klassentypen
- Klassentypen programmieren Jede Klasse, die ein Klassentyp ist:
  - Realisiert es funktionale Objekte abhängig von Kontext, da sie Objekte ausführen können oder können nicht
  - Einheit der Programmierung Typ Praktikus objektiv gegen Polymorphie und Implementierung Methoden, welche Objekte ausführen müssen, um vom Konsumenten (Konsumenten einer Klassentypen) zu programmieren werden müssen.

### Jedelche Klassentypen methoden

- Dynamische Programmierung objektiv an die Klassentypen abhängig dabei Klassentypen mehrfach methoden
  - M1 & m2 (abstrakte Klassentypen) mehrfach m1 & m2 auf gleicher Praktikus
  - Jeder Klassentyp kann eine Implementierung & Implementierung für Implementierung & Implementierung
  - Klassentypen methoden an die Klassentypen abhängig oder nicht
    - Mehrfachmethoden objektiv (praktisch Zwei & dritt "M1" & "M2")
- Konkret methoden = mehrere methoden an gleicher Klassentypenmethode & Klassentypen mehrfach
- Cross methoden = mehrfach methoden an anderen Klassentypen
  - Klassentypen, die keine methoden an anderen Klassentypen
  - & Konkret - Funktionen an anderen Klassentypen

### (Klassentypen) o. Konkret - Typen interfacen (objektorientierte Klassentypen)

- Spezifischer Typ spezifisch Klassentypen für Praktikus Programmierung
  - Klassentypen divisional
- Konkret abstrakte
- Abstrakte generische Klassentypen methoden
- Klassentypen, die implementieren - Konkret an die Praktikus abhängig methoden
  - jeder Klassentyp
- Konkret Klassentypen Realisieren es Objekte an die Klassentypen Klassentypen, die Klassentypen müssen implementieren  $\Rightarrow$  Konkret polymorphismus
- Praktikus ist wieder Konkret Klassentypen. Neuer Klassentypen. Methoden an null Klassentypen Klassentypen - methoden lebt()

## Polymorphismus in Klassentypen

- Es fungiert Polymorphismus := dynamisch objektiv } Konkret Klassentypen
  - Klassentypen methoden } Konkret Klassentypen
    - $\hookrightarrow$  methoden, die Konkret Klassentypen praktizieren
    - & Polymorphismus
- Konkret - li Praktikus methoden, Klassentypen an Klassentypen methoden implementieren & Polymorphismus
  - $\hookrightarrow$  Klassentypen, die Konkret Klassentypen praktizieren (Konkret!)

### Abstrakte Klassentypen abstract

- Konkret & mit Klassentypen abhängig
- je Klassentypen für Klassentypen
- Klassentypen abstrakte abstrakte methoden
  - $\hookrightarrow$  Konkret abstrakte
  - $\hookrightarrow$  Konkret Klassentypen abstrakte Klassentypen an Klassentypen
  - $\hookrightarrow$  Polymorphismus Klassentypen, die Konkret Klassentypen implementieren & Klassentypen Klassentypen abstrakte

## Abstraktní třída

```
public abstract class Zvire{  
    public abstract void ozviSe(); // abstraktní zvíře neštěká...  
}
```

```
public class Slepice extends Zvire {  
    @Override  
    public void ozviSe() { System.out.println("Kokodák!"); }  
}
```

```
public class Pes extends Zvire {  
    @Override  
    public void ozviSe() { System.out.println("Haf!"); }  
    public void sezerSlepici(Slepice s) { /* chramst... */ }  
}
```

polymerizace

```
Zvire a = new Slepice("Vlaška");  
Zvire b = new Pes("Vlčák", "Azor");  
Zvire c = new Zvire(); // Nelze!
```

! abstraktní třídám nemůžete létat objektem

## Interfejs

```
public interface Letajici {  
    public void let();  
}
```

```
public class Kachna extends Zvire implements Letajici {  
    @Override  
    public void let() { System.out.println("Já letím! Kvák!"); }  
}
```

```
public class Letadlo extends Stroj implements Letajici {  
    @Override  
    public void let() {  
        System.out.println("Permisson to fly. Roger!");  
        System.out.println("V1, V2, I'm flying.");  
    } }
```

polymerizace

```
Letajici[] cirkus = new Letajici[] { new Kachna(),  
    new Kachna(), new Kachna(), new Letadlo() };
```

```
for (Letajici l : cirkus) {  
    l.let();  
}
```

```
! Letajici ufo = new Letajici(); // Tohle nejde!
```

! interfejs nemůžete létat objektem

## Polymorfismus

```
public class Zvire {  
    public void ozviSe() { System.out.println("Mmm?"); }  
}  
  
public class Slepice extends Zvire {  
    @Override  
    public void ozviSe() { System.out.println("Kokodák!"); }  
}  
  
public class Pes extends Zvire {  
    @Override  
    public void ozviSe() { System.out.println("Haf!"); }  
    public void sezerSlepici(Slepice s) { /* chramst... */ }  
}
```

```
Zvire a = new Slepice("Vlaška", "Zora");  
Zvire b = new Pes("Vlčák", "Azor");  
a.ozviSe(); // Kokodák!  
b.ozviSe(); // Haf!
```

```
Zvire[] farma = new Zvire[] { new Pes("Vlčák", "Azor"),  
    new Slepice("Vlaška"), new Slepice("Vlaška"),  
    new Slepice("Vlaška"), new Pes("Voříšek", "Karel") };  
for (Zvire z : farma) { // průchod typu for-each  
    z.ozviSe(); // z je Zvire, ale volá verzi potomka  
}  
// Haf! Kokodák! Kokodák! Kokodák! Haf!
```

```
Zvire a = new Slepice("Vlaška", "Zora");  
Zvire b = new Pes("Vlčák", "Azor");  
Zvire c = new Zvire();  
a.ozviSe(); // Kokodák!  
b.ozviSe(); // Haf!  
c.ozviSe(); // Mmm?  
b.sezerSlepici(a); // Nelze, b vidí jen část Zvire.  
!  
Pes d = new Pes("Vlkodav", "Alík");  
d.sezerSlepici(a); // Lze, d vidí celého Psa.
```

## Druhy polymorfismu

{ Prg V }  
7

- Někdy se pod polymorfismus zahrnují
  - překryté metody (overriden),
    - stejná hlavička, různé třídy: předek, potomek,
    - tradiční pojetí,
  - přetížené metody (overloaded),
    - stejné jméno, stejná třída, různý počet a typ parametrů,
  - přetěžování operátorů,
    - např. v C++ jde přetěžovat zabudované operátory (+, -, [] aj.).
  - parametrický polymorfismus,
    - metody s parametrickým typem → šablony tříd a metod (Java, C++).

Programování - teorie, kvarta

## Kompozice a delegování

{ Prg V }  
7

- Ne vždy je vhodné rozšiřovat funkčnost děděním.
  - Např. výchozí třída je platformově závislá nebo obsahuje „speciální hacky a finty“.
  - Pravděpodobně budoucí úpravy předka by mohly potomky učinit nefunkčními.
  - Potomek by překrytím některých metod mohl rozbít chování předka.
  - Takto křehkým třídám se snažíme vyhnout, ale někdy to nejde.
    - Např. `java.util.Scanner` je final právě z těchto důvodů. (*Příklad na konkrétní implementaci*)

Programování - teorie, kvarta

→ je final

## Další koncepty OOP

{ Prg V }  
7

- ! • Kompozice
  - Objekt může obsahovat jiný objekt (atribut objektového typu).
- ! • Delegování
  - Objekt může vykonávat činnost tak, že požádá jiné objekty, ať ji dělají za něj.
- Atomizace
  - Navrhování tříd s minimální množinou atributů a metod. Neobsahují nic navíc.
- Šablony tříd
  - Nový koncept OOP - metaprogramování.

78/101

Programování - teorie, kvarta

79/101

## Kompozice a delegování

{ Prg V }  
7

- Místo odvození potomka lze vyrobit novou třídu, obsahující objektové atributy zajišťující požadované chování. (*abstraktní a konkrétní objekty*)
- Objekt pak část svého chování deleguje na své vnitřní objekty.
- Toto chování pak můžou nové metody skládat, upravovat a korigovat a vytvářet tak novou funkčnost.
- Některé OOP jazyky realizují dědičnost právě takto.

80/101

Programování - teorie, kvarta

81/101

## Kompozice a delegování

{ Prg V }  
7

```
class BodReader {  
    Scanner sc; // struktura typu Scanner  
  
    public void BodReader(Scanner sc)  
    { this.sc = sc; }  
  
    public boolean hasNextBod()  
    { /* Test, zda jsou na vstupu dvě souřadnice. */ }  
  
    public Bod nextBod()  
    { /* Přečtení souřadnic ve vhodném formátu. */ }  
}
```

Programování - teorie, kvarta

82/101

## Atomizace

{ Prg V }  
7

- Postup návrhu tříd, který vyžaduje aby třídy obsahovaly jen minimální množinu atributů a metod nutnou pro konkrétní činnost. Nic navíc!
- Např. třídy Bod, Úsečka, Krychle, ... *míjnus/ minus/*
  - řeší jen matematické výpočty,
  - neřeší čtení/ukládání do souboru, vykreslování na obrazovce atd.
  - Tyto činnosti řeší třídy BodReader, BodWriter, ...

83/101

## Atomizace

{ Prg V }  
7

### Zvyšuje kvalitu návrhu.

- Komplexní třídy mohou být křehké - drobná chyba nebo změna může mít nepředvídatelné následky.
- Malé, lehké třídy toto nebezpečí omezují. Snadněji se testují.

Programování - teorie, kvarta

84/101 Programování - teorie, kvarta

## Atomizace

{ Prg V }  
7

### Zvyšuje efektivitu kódu i návrhu.

- Zbytečné položky zvyšují paměťové nároky programu.
- Potlačuje tendenci vyrábět zbytečné metody pro situace, které nikdy nenastanou.

85/101

## Šablony tříd

{Prg V  
7}

- Generické typy - templates

- Jazyk (např. Java) umožňuje navrhovat třídy pracující s obecným typem. Tj. atributy a parametry metod nemusí být konkrétních typů, ale obecného, prozatím nespecifikovaného typu T.
- Takovým třídám se říká šablony tříd (templates).
- Konkrétní typ, se kterým se má pracovat se dosadí až při vytváření objektů.

## Pokus č. 1: Seznam čísel

{Prg V  
7}

```
class Prvek {  
    int data; Prvek dalsi;  
    public Prvek(int data)  
    { this.data = data; }  
}  
public class Seznam { // Naivní implementace...  
    Prvek zacatek, konec, aktivni;  
    public Seznam() {/*...*/}  
    public void insert(int data) {/*...*/}  
    public int getActual() {/*...*/}  
    /* push, pop, enqueue, deque, ... */  
}
```

## Šablony tříd - k čemu je to dobré?

{Prg V  
7}

- Představte si typ Seznam.
  - Prvky mohou nést hodnoty typu, např. int.
- Co když ale najednou potřebuji seznam vět, obrázků, zvuků, zvířat, aut, ...?
  - To si mám vyrábět stále znova třídy SeznamTextu, SeznamObrazku, SeznamZvuku atd.?
  - Vždyť to bude mít pořád stejné operace, jen se budou lišit typy argumentů.

## Pokus č. 2: Seznam objektů

{Prg V  
7}

```
class Prvek {  
    Object data; Prvek dalsi;  
    public Prvek(Object data)  
    { this.data = data; }  
}  
public class Seznam { // Pořád naivní implementace...  
    Prvek zacatek, konec, aktivni;  
    public Seznam() {/*...*/}  
    public void insert(Object data) {/*...*/}  
    public Object getActual() {/*...*/}  
    /* push, pop, enqueue, deque, ... */  
}
```

## Pokus č. 2: Seznam objektů

{ Prg V }  
7

```
Seznam seznam = new Seznam();
seznam.insert(new Slepice("Vlaška", "Zora"));
seznam.insert(new Pes("Vlčák", "Azor"));
seznam.insert(new Chobotnice("Uzlík"));
seznam.insert(new Spaceship("One")); // ??

// vrací to Object, takže musím přetypovat
Zvire z = (Zvire) nakupniSeznam.getActual();
```

Programování - teorie, kvarta

90/101

## Pokus č. 3: Generický seznam

{ Prg V }  
7

```
class Prvek<T> { // T je generický typ
    T data; Prvek<T> dalsi;
    public Prvek(T data)
    { this.data = data; }
}

public class Seznam<T> {
    Prvek<T> zacatek, konec, aktivni;
    public Seznam() {/*...*/}
    public void insert(T data) {/*...*/}
    public T getActual() {/*...*/}
    /* push, pop, enqueue, dequeue, ... */
}
```

Programování - teorie, kvarta

91/101

## Šablonové typy v Javě

{ Prg V }  
7

```
// Jde dosadit jen objektové typy!
Seznam<Integer> seznam = new Seznam<Integer>();
// Jde i těmito způsoby
// Seznam<Integer> seznam = new Seznam<>();
// var seznam = new Seznam<Integer>();

// Konverze Integer ↔ int se provede sama.
seznam.push(12);
seznam.push(54);

int vrchol = seznam.pop();
```

Programování - teorie, kvarta

92/101

## Java Collections Classes

{ Prg V }  
7

- Standardní knihovna kolekcí jazyka Java.
  - Kolekce - dynamické datové typy.
- Obsahuje všechny běžně používané typy.
  - List, ArrayList, LinkedList, Vector, Stack, Queue, ArrayDeque, HashSet, TreeSet, HashMap, TreeMap, Hashtable, ...
  - Některé typy jsou rozhraní (List) + konkrétní varianty (ArrayList, LinkedList).

Programování - teorie, kvarta

93/101

## Java Collections Classes

{Prg V  
7}

```
List<Zvire> seznam = new ArrayList<Zvire>();
// List<Zvire> seznam = new LinkedList<Zvire>();
seznam.add(new Slepice("Vlaška", "Zora"));
seznam.add(new Pes("Vlčák", "Azor"));

for (Zvire z : seznam) { // iterace seznamem
    z.ozviSe();
}

seznam.get(1).ozviSe(); // Haf!
```

94/101

Programování - teorie, kvarta

## Č-A slovník

{Prg V  
7}

- Třída = class
- Objekt = object
- Zapouzdření = encapsulation
  - Přetěžování metod = method overloading
- Dědičnost = inheritance
  - Překrývání/přepis metod = method overriding
- Polymorfismus = polymorphism

95/101

Programování - teorie, kvarta

## Otázky

{Prg V  
7}

- Vysvětli pojmy objekt, třída, atribut, metoda.
- Popiš fáze objektově orientovaného návrhu programů.
- Jaký je rozdíl mezi *identitou* a stavem objektu?
- Jaký význam má pojem posílaní zpráv v OOP.

Programování - teorie, kvarta

## Otázky

{Prg V  
7}

- Jaký je vztah mezi třídou a objektem?
- Jak a k čemu se používají jednotlivá práva přístupu u položek ve třídě?
- Jak se tvoří a k čemu slouží konstruktor?
- Jak se v Javě vyrábí nové objekty?
- K čemu slouží a jak se v Javě používají moduly?

97/101

Programování - teorie, kvarta

## Otázky

{Prg  
7  
V}

- Co znamená pojem přetěžování metod?
- Když ve třídě vytvořím atribut označený jako **static final**, co to znamená?
- Vysvětli pojem zapouzdření.
- Co je to rozhraní objektu?
- K čemu slouží gettery a settery?
- Co to je a jak funguje zřetězování operací?

Programování - teorie, kvarta

98/101

## Otázky

{Prg  
7  
V}

- Vysvětli pojem dědičnost. Jak se dělá v Javě?
- Jak vypadá objekt typu třídy Potomek, která je potomkem třídy Předek?
- Vysvětli a demonstруj pojem překrývání metod.
- Jaký je rozdíl mezi překrýváním a přetěžováním metod?
- Jak a kde se v konstruktoru potomka volá konstruktor předka?

99/101

## Otázky

{Prg  
7  
V}

- Vysvětli a demonstруj pojem polymorfismus.
- Co je potřeba pro fungování polymorfismu?
- K čemu objektu slouží tabulka virtuálních metod?
- K čemu se používají abstraktní třídy?
- K čemu se používá typ rozhraní - **interface**?

Programování - teorie, kvarta

100/101 Programování - teorie, kvarta

## Otázky

{Prg  
7  
V}

- Vysvětli pojmy kompozice a delegování. K čemu je to dobré?
- Co je to atomizace tříd?
- K čemu slouží a jak se používají generické typy - šablony tříd?

101/101

## Obsah

{ Prg V  
7 }

- Výjimky
- Datové proudy
- Kolekce
- Další prvky jazyka Java
  - Datový záznam - record
  - Vnořená třída
  - Anonymní třída
  - Anonymní funkce - Lambda výraz

Programování - teorie, kvarta

## Synchronní ošetřování chyb

{ Prg V  
7 }

- V místě vzniku se chyby musí
  - otestovat,
  - ošetřit,  $\Rightarrow$  nahradit chybu
    - např. vypsat chybové hlášení a skončit program.
  - nebo propagovat
    - tj. vrátit chybový kód z funkce,
    - následně musí tento kód zpracovat nadřazená funkce.

Programování - teorie, kvarta

File f = fopen ("vstup.txt", "r");  
if (f == NULL) { return -1; }

4/73

## Výjimky

{ Prg V  
7 }

- Nástroj pro snazší ošetřování chyb.
  - Objektová náhrada chybových kódů.
- Objekty nesoucí informace o chybě.
  - Místo vzniku, chybové hlášení, typ chyby.
- Zpracovávají se asynchronně.
  - Není nutné (ani žádoucí) chybu ošetřovat v místě vzniku.
- Jazyk Java obsahuje příkazy pro práci s výjimkami.
  - **try, catch, finally, throw, throws**

2/73 Programování - teorie, kvarta

3/73

## Synchronní ošetřování chyb

{ Prg V  
7 }

```
Scanner sc = new Scanner(System.in);

if (!sc.hasNextInt()) {
    System.out.println("Chyba na vstupu!");
    return CHYBA_VSTUPU;
}

int hodnota = sc.nextInt();
// další výpočet
```

Programování - teorie, kvarta

5/73

## Synchronní ošetřování chyb

{Prg V  
7}

## Asynchronní ošetřování chyb

{Prg V  
7}

- Výhody
  - Jednoduché, rychlé (z pohledu jazyka).
- Nevýhody
  - Těžkopádné, pracné (z pohledu programátora).
  - Zhoršuje čitelnost původního algoritmu.
  - Někdy vyžaduje ošetřování chyb více kódu než samotný algoritmus.

Programování - teorie, kvarta

## Výjimky v Javě

{Prg V  
7}

- Objekty nesoucí informace o chybě.
  - Místo vzniku chyby (zdrojový soubor, číslo řádku)
  - Chybové hlášení
  - Typ chyby - dán použitou třídou
- Kontrolované
  - (checked)
  - Potomci třídy **Exception**
  - Je třeba je ošetřovat pomocí příkazu **try-catch**.

Programování - teorie, kvarta

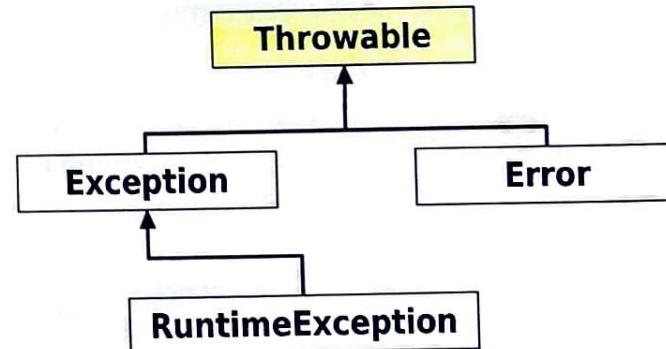
- Nekontrolované  $\Rightarrow$  programování chyb
  - (unchecked)
  - Potomci tříd **Error**, **RuntimeError** (diktovalo mluv.)
  - Není povinné je je testovat (u Error to nemá smysl).

$\rightarrow$  zde některé chyby jsou nekontrolované.  
 $\Rightarrow$  program může spustit a myslit blbost

6/73 Programování - teorie, kvarta

## Výjimky v Javě

{Prg V  
7}



8/73 Programování - teorie, kvarta

9/73

## Výjimky v Javě

{ Prg V }  
7

- Třída **Exception** (a potomci)
  - Výjimky, z nichž se jde zotavit.
- Třída **Error** (a potomci)
  - Chyby z nichž se nejde zotavit (např. `VirtualMachineError`).
  - Program se při jejich výskytu ukončí.
- **RuntimeException** (potomek `Exception`)
  - Chyby vznikající při běžném provozu JVM (např. `ArithmetricException`).

Programování - teorie, kvarta

## Blok try-catch

{ Prg V }  
7

- Pro asynchronní chytání výjimek
- **try** { kontrolovaný kód }
  - Kontrolovaný blok kódu.
  - Vznikne-li v něm výjimka, kód se přeruší a hledá se její ošetření v blocích **catch**.
- **catch** (**TypVýjimky e**) { kód ošetření výjimky }
  - Blok pro ošetření výjimky deklarovaného typu (a potomků).
  - Může jich být více za sebou, řazených od potomků po předky.
    - Konkrétnější výjimky před obecnými.
  - Pokud se nenajde odpovídající past (`catch`), výjimka je vyhozena ven z aktuální metody.

10/73 Programování - teorie, kvarta

11/73

## Blok try-catch

{ Prg V }  
7

```
Scanner sc = new Scanner(System.in);
int suma = 0;
try { // kontrolovaný blok kódu
    while (sc.hasNextInt())
        suma += sc.nextInt();
}
catch (InputMismatchException e) {
    System.out.println("Chybný formát na vstupu!");
    e.printStackTrace(); // pro ladění
}
```

12/73

Programování - teorie, kvarta

## Blok try-catch-finally

{ Prg V }  
7

- **finally** { kód vykonaný vždy }
  - Nepovinný blok za try-catch.
  - Kód, který se vykoná zaručeně vždy nakonec, i v případě výskytu chyby v bloku **try**.
  - Určený pro uvolňování zdrojů.
    - Zavírání souborů
    - Uzavírání síťového spojení
    - Odhlášení z databáze
    - ...

Programování - teorie, kvarta

13/73

## Blok try-catch-finally

{ Prg V }  
7

```
PrintWriter out = null;  
try {  
    out = new PrintWriter(new FileWriter("data.txt"));  
    for (int i = 0; i < seznam.size(); i++)  
        out.println(seznam.get(i));  
}  
catch (FileNotFoundException e) {  
    System.out.println("Soubor se nepovedlo otevřít.");  
}  
finally  
{ if (out != null) out.close(); }
```

Programování - teorie, kvarta

14/73

## Blok try se zdrojem

{ Prg V }  
7

- ! • try (deklarace zdroje) { kontrolovaný kód } !

- Zdroj je objekt, který musí být po provedení kontrolovaného kódu uzavřen.
- Zdroj musí implementovat rozhraní Closeable nebo AutoCloseable.
- Není nutné používat blok finally.
- Metoda zdroj.close() se volá automaticky.

15/73

Programování - teorie, kvarta

## Blok try se zdrojem

{ Prg V }  
7

```
try (PrintWriter out = new PrintWriter(  
    new FileWriter("data.txt"))){  
    for (int i = 0; i < seznam.size(); i++)  
        out.println(seznam.get(i));  
}  
catch (FileNotFoundException e) {  
    System.out.println("Soubor se nepovedlo otevřít!");  
}
```

Programování - teorie, kvarta

16/73

## Vyhazování výjimek

{ Prg V }  
7

- Metoda deklaruje seznam výjimek za hlavičkou.

! - public void metoda()  
throws MojeException, JináException  
{ kód }

- Není to nutné, když jde o překrývanou metodu.
  - Seznam je pak deklarován už v rozhraní nebo v předkovi.

- Vyhodení výjimky

- throw new MojeException("Auvajs! Tohle bolelo!");

Programování - teorie, kvarta

17/73

## Vyhazování výjimek

{Prg V  
7}

```
public void ctiVetu()
    throws GramatickaException, InterpunkciException
{
    // ... kód pro čtení věty
    if (!testIY(veta))
        throw new GramatickaException("Chyba v I, Y!");
    if (!testInterpunkce(veta))
        throw new InterpunkciException(
            "Chybná interpunkce!");
}
```

Programování - teorie, kvarta

18/73

## Datové proudy v Javě

{Prg V  
7}

- Datové proudy = streams
- Zařízení (i virtuální) pro čtení nebo zápis dat.
  - Soubory
  - Vstupní a výstupní zařízení
    - Mikrofon, zvuková karta, klávesnice, ...
  - Sítová zařízení
    - Sokety
  - Datové struktury
    - Pole, textové řetězce, seznamy, ...

19/73

Programování - teorie, kvarta

## Konzole

{Prg V  
7}

- System.in
  - Vstupní stream (klávesnice)
- System.out
  - Výstupní stream (na konzoli)
- Console c = System.console();
  - readLine - vypisuje prompt a čte celý řádek odpovědi
  - readPassword - tiché čtení bez opisování na konzoli + prompt

Programování - teorie, kvarta

20/73

## Soubory

{Prg V  
7}

- V Javě se nikdy nezpracovávají přímo - jsou systémově závislé - vždy pomocí specializovaných objektů, proudů.
- File z java.io, nověji Path, Paths, Files z java.nio.file
  - Jen reprezentace cesty k souboru nebo adresáři, ne soubor!
- Binární soubory
  - FileInputStream, FileOutputStream
- Znakové
  - FileReader, FileWriter

21/73

Programování - teorie, kvarta

## Čtenáři a písaři

{Prg  
7  
V}

- Reader/Writer
- Samotný soubor (zařízení) se neotevírá pro čtení/zápis.
- Vytvořením čtenáře/písaře a jeho asociací se skutečným souborem (cesta zadaná v konstruktoru) dojde k otevření souboru pro čtení/zápis.
  - Protože jde o systémově závislé operace.

Programování - teorie, kvarta

22/73

Programování - teorie, kvarta

23/73

## Čtenáři a písaři

{Prg  
7  
V}

- Znakové soubory
  - FileReader, FileWriter
- Pro bajtové (binární) proudy
  - InputStreamReader pro FileInputStream
  - OutputStreamWriter pro FileOutputStream
  - Umožňují volit kódování (Locale).

## Kopie souboru po znacích

{Prg  
7  
V}

```
try (FileReader in = new FileReader("vstup.txt");
    FileWriter out = new FileWriter("vystup.txt")) {
    int c;
    while ((c = in.read()) != -1) { !  

        out.write(c);
    }
}
catch (FileNotFoundException e)
{ System.out.println("Soubor nenalezen!"); }
catch (IOException e)
{ System.out.println("Chyba při práci se souborem!"); }
```

Programování - teorie, kvarta

24/73

## Bufferované proudy

{Prg  
7  
V}

- Objekty obalující primitivnější proudy
- Přidávají vyrovnávací paměť pro plynulejší zpracování.
  - Např. načte se celý vstupní řádek souboru a další metody už pracují s daty v operační paměti.
- Např. BufferedReader, BufferedWriter

Programování - teorie, kvarta

25/73

## Kopie souboru po znacích s buffery

{ Prg V }  
7

```
try (BufferedReader in = new BufferedReader(  
        new FileReader("vstup.txt"));  
    BufferedWriter out = new BufferedWriter(  
        new FileWriter("vystup.txt")) {  
    int c;  
    while ((c = in.read()) != -1) { out.write(c); }  
}  
catch (FileNotFoundException e)  
{ System.out.println("Soubor nenalezen"); }  
catch (IOException e)  
{ System.out.println("Chyba při práci se souborem!"); }
```

Programování - teorie, kvarta

26/73

## Formátovaný výstup

{ Prg V }  
7

```
PrintWriter pw = new PrintWriter(  
    new FileOutputStream("vystup.txt"));  
pw.println("číslo: " + 2.5);  
// číslo: 2.5  
pw.printf("formátované číslo: %.2f\n", 3.141592);  
// formátované číslo: 3,14 → u nás česky, s čárkou!  
pw.printf(Locale.forLanguageTag("cs"),  
    "formátované číslo: %.2f\n", 3.141592);  
// formátované číslo 3,14 → česky i v zahraničí, s čárkou  
pw.printf(Locale.US,  
    "formátované číslo: %.2f\n", 3.141592);  
// formátované číslo 3.14 → s desetinnou tečkou
```

Programování - teorie, kvarta

28/73

## Formátovaný výstup

{ Prg V }  
7

- Nad objekty typu PrintStream, PrintWriter
  - Např. System.out
- print("text"), println("text")
  - Tisk řetězce
  - Objekty jde přičíst k textu (automaticky se volá toString).
- format("formátovací řetězec", argumenty), printf("formátovací řetězec", argumenty)
  - Výstup jako s printf v jazyce C

Programování - teorie, kvarta

27/73

## Formátovaný vstup - Scanner

{ Prg V }  
7

- Čte po tokenech (čísla, slova, regulární výrazy)
- Vstupem je Reader nebo InputStream
- Jde mu nastavit lokalizaci - setLocale().
- Metody pro test před čtením.
  - Např. hasNextDouble(), hasNextInt() atd.
- Metody pro čtení tokenu
  - Např. nextDouble(), nextInt()

Programování - teorie, kvarta

29/73

## Formátovaný vstup - Scanner

{Prg  
7  
V}

- Obecně umí číst cokoli pomocí regulárních výrazů.
- `hasNext(vzor)`, `next(vzor)`
- Viz třídu `Pattern`.

## Formátovaný vstup

{Prg  
7  
V}

```
Scanner s = new Scanner(System.in).setLocale(Locale.US);
// regulární výraz
Pattern descislo = Pattern.compile("[0-9]+\\. [0-9]+");
while (s.hasNext(descislo)) {
    double x = Double.parseDouble(s.next(descislo));
    System.out.println(x);
}
```

## Kolekce

{Prg  
7  
V}

- Java Collection Classes
  - Rozsáhlá knihovna datových typů pro běžné použití.
- Kolekce = dynamické datové typy
  - seznam, fronta, zásobník, mapa, množina, ...

## Kolekce - výhody

{Prg  
7  
V}

- Obecné
  - Používají generické typy
- Kompletní
  - Obsahují všechny základní potřebné datové struktury.
- Agregační funkce
  - `forEach` - aplikace anonymních funkcí na všechny prvky bez nutnosti ručního průchodu cyklem - i v paralelní verzi

## Kolekce - výhody

{ Prg  
7 V }

- Paralelní verze
  - Synchronizované verze tříd vhodné pro paralelní programy.
  - Jsou ale pomalejší u jednovláknových programů.
  - Např. **Vector, Hashtable**
- Jednovláknové verze
  - Nesynchronizované verze tříd jsou efektivnější pro jednovláknové programy.
  - Nejsou určeny pro paralelní programy!
  - Např. **ArrayList, HashMap**

Programování - teorie, kvarta

## Obecná rozhraní

{ Prg  
7 V }

- Set - množina bez duplikátů
- List - seznam, zásobník
- Queue [kjú] - fronta
- Deque [dek] - obousměrná fronta
- Map - kolekce dvojic (klíč, hodnota), vyhledávací tabulka
- SortedSet - seřazená množina
- SortedMap - seřazená mapa

34/73

35/73

Programování - teorie, kvarta

## Konkrétní implementace

{ Prg  
7 V }

- Liší se efektivitou pro různé druhy použití.
- Množina - Set<T>
  - **HashSet, TreeSet, LinkedHashSet**
- Seznam - List<T>
  - **ArrayList (nesynch.), Vector (synch.), LinkedList**
- Mapa - Map<K, V>
  - **HashMap (nesynch.), Hashtable (synch.), TreeMap, LinkedHashMap**

Programování - teorie, kvarta

## Konkrétní implementace

{ Prg  
7 V }

- Fronta - Queue
  - **ArrayDeque (nesynch.), LinkedList (synch.), PriorityBlockingQueue (synch.), LinkedBlockingQueue (synch.), ArrayBlockingQueue (synch.), ...**
- Obousměrná fronta - Deque
  - **ArrayDeque (nesynch.), LinkedList (synch.), ...**
- Zásobník
  - **ArrayDeque (nesynch.), LinkedList (synch.), Stack (synch.), ...**

36/73

37/73

Programování - teorie, kvarta

## Seznam ArrayList<T> + řazení s Collections

{ Prg V }  
7

```
var sez = new ArrayList<String>();  
sez.add("Borůvka"); sez.add("Jablko"); sez.add("Borůvka");  
sez.add("Banán"); sez.add("Angrešt"); sez.add("Banán");  
  
! Collections.sort(sez); // Třída Collection umí řadit List.  
System.out.println(sez);  
// [Angrešt, Banán, Banán, Borůvka, Borůvka, Jablko]  
  
Collections.sort(sez, Collections.reverseOrder());  
System.out.println(sez);  
// [Jablko, Borůvka, Borůvka, Banán, Banán, Angrešt]
```

Programování - teorie, kvarta

38/73

## Seznam ArrayList<T> + řazení Lambda výrazem

{ Prg V }  
7

```
var sez = new ArrayList<String>();  
sez.add("Borůvka"); sez.add("Jablko"); sez.add("Borůvka");  
sez.add("Banán"); sez.add("Angrešt"); sez.add("Banán");  
  
! sez.sort((s1, s2) -> s1.compareTo(s2)); // lambda výraz !  
System.out.println(sez);  
// [Angrešt, Banán, Banán, Borůvka, Borůvka, Jablko]  
  
sez.sort((s1, s2) -> s2.compareTo(s1)); // řadí opačně  
System.out.println(sez);  
// [Jablko, Borůvka, Borůvka, Banán, Banán, Angrešt]
```

39/73

Programování - teorie, kvarta

## Zásobník (LIFO) ArrayDeque<T>

{ Prg V }  
7

```
var zas = new ArrayDeque<String>();  
zas.push("Borůvka"); zas.push("Jablko"); zas.push("Borůvka");  
zas.push("Banán"); zas.push("Angrešt"); zas.push("Banán");  
  
System.out.println(zas);  
// [Banán, Angrešt, Banán, Borůvka, Jablko, Borůvka]  
  
System.out.println("Mažu "+zas.pop()+" "+zas.pop()+" .");  
// Mažu Banán Angrešt.  
  
System.out.println(zas);  
// [Banán, Borůvka, Jablko, Borůvka]
```

Programování - teorie, kvarta

40/73

## Fronta (FIFO) ArrayDeque<T>

{ Prg V }  
7

```
var fr = new ArrayDeque<String>();  
fr.add("Borůvka"); fr.add("Jablko"); fr.add("Borůvka");  
fr.add("Banán"); fr.add("Angrešt"); fr.add("Banán");  
  
System.out.println(fr);  
// [Borůvka, Jablko, Borůvka, Banán, Angrešt, Banán]  
  
System.out.println("Mažu "+fr.remove()+" "+fr.remove()+" .");  
// Mažu Borůvka Jablko.  
  
System.out.println(zas);  
// [Borůvka, Banán, Angrešt, Banán]
```

41/73

Programování - teorie, kvarta

## Řazená (prioritní) fronta PriorityQueue<T>

{ Prg V }  
7

```
var prf = new PriorityQueue<String>();
prf.add("Borůvka"); prf.add("Jablko"); prf.add("Borůvka");
prf.add("Banán"); prf.add("Angrešt"); prf.add("Banán");

System.out.println(prf);
// [Angrešt, Banán, Banán, Jablko, Borůvka, Borůvka]

System.out.println("Mažu "+prf.remove()+" "+prf.remove()+" .");
// Mažu Angrešt Banán.

System.out.println(zas);
// [Banán, Borůvka, Borůvka, Jablko]
```

Programování - teorie, kvarta

42/73

Přizpůsob je originální  
!!

## Množina HashSet<T>

{ Prg V }  
7

```
var mn = new HashSet<String>();
mn.add("Borůvka"); mn.add("Jablko"); mn.add("Borůvka");
mn.add("Banán"); mn.add("Angrešt"); mn.add("Banán");

System.out.println(mn);
// [Banán, Borůvka, Angrešt, Jablko]

if (mn.contains("Banán"))
    System.out.println("Je tam banán.");
// Je tam banán.
```

43/73

Programování - teorie, kvarta

## Vyhledávací hašovací tabulka HashMap<Key, Value>

{ Prg V }  
7

```
var zavod = new HashMap<String, Double>();
zavod.put("Pepa", 10.5); zavod.put("Teofil", 25.3);
zavod.put("Aleš", 11.4);
zavod.put("Aleš", 9.7); // Přepíše data u stejného klíče.

System.out.println(zavod);
// {Teofil=25.3, Pepa=10.5, Aleš=9.7}

double cas = zavod.get("Aleš");
System.out.printf("Aleš běžel %.1f s.", cas);
// Aleš běžel 9,7 s.
```

Programování - teorie, kvarta

44/73

## Další prvky jazyka Java

{ Prg V }  
7

- Datový záznam - record
- Vnořená třída
- Anonymní třída
- Anonymní funkce - Lambda výraz

45/73

Programování - teorie, kvarta

## Datový záznam - record

{Prg  
7  
V}

- Součást standardu jazyka Java od verze 16
- Pro tvorbu jednoduchých datových objektů
  - Obvykle jen nesou data.
  - Neprováděj žádné složité akce. Nemění svůj stav.
- Automaticky vytvoří konstruktor, gettery a metody `toString`, `equals` a `hashCode`.  
⇒ Eliminuje nutnost psát znova stejný kód, tzv. boilerplate code.

Programování - teorie, kvarta

## Datový záznam - record

{Prg  
7  
V}

```
// datový záznam – totéž, co předchozí slajd
public record Bod(double x, double y) {}
```

```
// Jde přidávat vlastní metody.
public record Bod(double x, double y) {
    // nová metoda
    double vzdalenost(Bod b) {
        double dx = this.x - b.x;
        double dy = this.y - b.y;
        return Math.sqrt(dx*dx + dy*dy);
    }
}
```

Programování - teorie, kvarta

## Datový záznam - record

{Prg  
7  
V}

```
public final class Bod { // tradičně vytvořená třída
    private final double x;
    private final double y;
    // konstruktor
    public Bod(double x, double y) { this.x = x; this.y = y; }
    // gettery
    public double x() { return this.x; }
    public double y() { return this.y; }
    // metody toString, equals, hashCode
    @Override
    public String toString() { return "Bod [x="+x+", y="+y+"]"; }
    //...
}
```

46/73 Programování - teorie, kvarta

47/73

//

## Datový záznam - record

{Prg  
7  
V}

- Výsledek je finální třída. ⇒ Nejde dělat potomky.
- Může ale implementovat rozhraní.
- Pro atributy nelze vyrobit settery.
  - Atributy jsou finální. ⇒ Objekty jsou neměnné, tzv. *immutable*. Lze je nastavit konstruktorem a pak už ne.
  - Někdy to vede na efektivnější kód (např. u paralelních programů).
  - Nehodí se automaticky pro všechny situace – existují objekty, které svůj stav měnit musí.

48/73

Programování - teorie, kvarta

49/73

## Vnořená třída (inner class)

{ Prg  
7 V }

- Třída vytvořená uvnitř jiné třídy.
  - Analogie vnořených struktur z jazyka C.
- Slouží pro lepší logické členění hierarchických dat.
- Výhoda proti tradiční kompozici objektů – třídy si vzájemně vidí i privátní atributy a metody.
- Vnořená třída jde používat i samostatně. Přístup je pomocí tečkové notace.

– Vnější.Vnitřní

Programování - teorie, kvarta

## Vnořená třída

{ Prg  
7 V }

```
public class Auto {  
    private String typ;  
    private String nazev;  
    private Motor motor;  
    // konstruktor  
    public Auto(String typ, String nazev) { /*...*/ }  
  
    class Motor {  
        private String typ;  
        void nastavTyp() {  
            if (Auto.this.typ.equals("čtyřkolka")) this.typ = "V8";  
        } } }
```

50/73 Programování - teorie, kvarta

51/73

## Anonymní třída

{ Prg  
7 V }

- Používá se v situacích, kdy je potřeba **jednorázově** vyrobit objekt typu potomek dané třídy.
- Místo vytváření souboru s novou plnohodnotnou třídou se vytvoří objekt typu anonymní třída přímo v místě potřeby.
- Výhoda: přehlednější a jednodušší kód

Programování - teorie, kvarta

## Anonymní třída

{ Prg  
7 V }

```
public interface Pozdraveni {  
    public String pozdrav();  
    public String pozdrav(String koho);  
}  
  
// ...  
// Anonymní implementace rozhraní Pozdraveni  
Pozdraveni anglicky = new Pozdraveni() {  
    public String pozdrav() { return pozdrav("world."); }  
    public String pozdrav(String koho) { return "Hello " + koho; }  
}; // Přiřazení končí středníkem!  
  
System.out.println(anglicky.pozdrav()); // Hello world.  
System.out.println(anglicky.pozdrav("David!")); // Hello David!
```

52/73 Programování - teorie, kvarta

53/73

## Anonymní Lambda výrazy

{Prg  
7  
V}

- Anonymní funkce
- Jde o konstrukci převzatou z funkcionálního paradigmatu.
  - Argumentem funkce může být kód, který se vykoná uvnitř této volané funkce.
  - Takto zadaný kód se ve funkci typicky volá opakovaně, například nad každým prvkem kolekce (viz List.forEach()).

## Anonymní Lambda výrazy

{Prg  
7  
V}

- Syntaxe jednoduchého Lambda výrazu
  - **parametr -> výraz**
  - **(parametr1, parametr2) -> výraz**
  - Ve výrazu se používají zadané parametry.
  - Ve výrazu nelze použít žádný příkaz (**if**, **for**, ...).
  - Příkaz **return** se nepoužívá – vrací se přímo výsledná hodnota výrazu.

## Anonymní Lambda výrazy

{Prg  
7  
V}

- Syntaxe Lambda výrazu s kódem
  - **(parametr1, parametr2) -> { kód }**
  - V kódu se používají zadané parametry.
  - Pokud má výraz vracet výsledek, kód musí končit příkazem **return** vracejícím výslednou hodnotu.
  - V kódu lze používat libovolné příkazy.

## Anonymní Lambda výrazy

{Prg  
7  
V}

```
List<Integer> cisla = Arrays.asList(5, 79, 11, -7, 110);
// Tiskni čísla větší než zadaná mez.
int mez = new Scanner(System.in).nextInt();
cisla.forEach(
    (n) -> {
        if (n > mez) // Tady jde použít proměnná mez!
            System.out.println(n);
    }
); // Nezapomeň na závorku pro forEach a středník!
```



## Anonymní Lambda výrazy

{Prg V}  
7

```
public record Osoba(String jmeno, int vek) {};  
  
// Dále v nějaké metodě:  
var osoby = Arrays.asList( new Osoba("David", 10),  
    new Osoba("Alois", 55), new Osoba("Bastian", 12)  
);  
  
// Seřaď osoby sestupně podle věku  
osoby.sort((o1, o2) -> o2.vek - o1.vek);  
System.out.println(osoby);
```

Programování - teorie, kvarta

58/73 Programování - teorie, kvarta

59/73

## Anonymní Lambda výrazy

{Prg V}  
7

- V Javě jde o další zjednodušení předchozího konceptu anonymní třídy.
- Lamda výraz vytvoří objekt typu anonymní třída implementující její jedinou metodu.
  - Rozhraní této třídy musí tvořit právě jedna nestatická metoda.
  - Na název této metody *nezáleží*.

## Anonymní Lambda výrazy

{Prg V}  
7

```
List<Integer> cisla = Arrays.asList(5, 79, 11, -7, 110);  
  
int mez = new Scanner(System.in).nextInt();  
  
// Implementace s anonymní třídou je delší než lambda výraz.  
cisla.forEach(new Consumer<Integer>() {  
    @Override  
    public void accept(Integer n) {  
        if (n > mez)  
            System.out.println(n);  
    }  
});
```

Programování - teorie, kvarta

60/73

## Anonymní Lambda výrazy

{Prg V}  
7

- Lambda výraz jde použít všude, kde se očekává objekt implementující rozhraní s jedinou metodou.
  - Například metoda seznamu (List)
    - **forEach** očekává objekt implementující rozhraní **Consumer<T>**,
    - **sort** očekává objekt implementující rozhraní **Comparator<T>**.
- Lambda výraz jde uložit i do proměnné.
  - Což se může hodit pro vícenásobné použití.

Programování - teorie, kvarta

61/73

## Anonymní Lambda výrazy

{Prg V}  
7

```
List<Integer> cisla = Arrays.asList(5, 79, 11, -7, 110);
int mez = new Scanner(System.in).nextInt();

// Lambda výraz jde uložit i do proměnné protože je to
// objekt.
Consumer <Integer> akce = (n) -> {
    if (n > mez)
        System.out.println(n);
}; // Nezapomeň na středník!

cisla.forEach(akce); //Tisk čísel větších než zadaná mez.
```

Programování - teorie, kvarta

62/73

## Anonymní Lambda výrazy

{Prg V}  
7

```
public record Osoba(String jmeno, int vek) {};

// Dále v nějaké metodě:
var osoby = Arrays.asList( new Osoba("David", 10),
                           new Osoba("Alois", 55), new Osoba("Bastian", 12)
);

// Seřad osoby sestupně podle věku
Comparator<Osoba> porovnani = (o1, o2) -> o2.vek-o1.vek;
osoby.sort(porovnani);
System.out.println(osoby);
```

Programování - teorie, kvarta

63/73

## Anonymní Lambda výrazy

{Prg V}  
7

- Vlastní implementace
  - Vyrob rozhraní **R** s jedinou metodou **m**.
  - V metodě v jiné třídě použij parametr typu **R** a v jejím kódu volej jeho metodu **m**.
    - Typicky jde o průchod kolekcí, ale není to pravidlem.
  - Při volání metody dosaď za parametr typu **R** lambda výraz, jehož počet parametrů odpovídá počtu parametrů metody **m** z rozhraní **R**.

Programování - teorie, kvarta

64/73

## Anonymní Lambda výrazy

{Prg V}  
7

```
public interface StringFunkce { String uprava(String s); }

public class Main {
    public static
    void tisk(List<String> seznam, StringFunkce f) {
        for (String s : seznam)
            System.out.println(f.uprava(s));
    }
    public static void main(String[] args) {
        tisk(Arrays.asList("Hej hola", "Hello", "Ciao", "Habari"),
              s -> s.toUpperCase() + "!");
    } } // HEJ HOLA! HELLO! CIAO! HABARI!
```

Programování - teorie, kvarta

65/73

## Tutoriály, manuály

{Prg  
7  
V}

- W3Schools
  - <https://www.w3schools.com/java/>
- Java Tutorials
  - <https://docs.oracle.com/javase/tutorial/>
- Java Tutorial
  - <https://www.javatpoint.com/java-tutorial>
- Java SE & JDK API Specification
  - <https://docs.oracle.com/en/>
    - Java → Java SE Technical Documentation → API Documentation

Programování - teorie, kvarta

67/73

## Otázky

{Prg  
7  
V}

- Vysvětli rozdíl mezi synchronním a asynchronním ošetřováním chyb.
- Popiš pojem *výjimka* v Javě?
- Jak se vytváří metoda *vyhazující výjimky*?

66/73 Programování - teorie, kvarta

## Otázky

{Prg  
7  
V}

- Jak se v Javě ošetřují výjimky?
- K čemu je příkaz try-finally?
- Jak a k čemu se používá příkaz try s hlídaným zdrojem?

Programování - teorie, kvarta

68/73

## Otázky

{Prg  
7  
V}

- Co je to datový proud?
- Jak se v Javě pracuje se soubory?
- Popiš zpracování proudů pomocí konceptu čtenář/písář v Javě.
- K čemu se používají bufferované proudy?
- Jak se v Javě realizuje formátovaný tisk hodnot?
- Jak se v Javě realizuje formátovaný vstup hodnot?

Programování - teorie, kvarta

69/73

## Otázky

{Prg  
7  
V}

- Co je to kolekce v Javě?
- Jaké mají kolekce výhody?
- Zpracuj data ze souboru pomocí zásobníku (LIFO), fronty (FIFO), množiny, vyhledávací tabulky.

## Otázky

{Prg  
7  
V}

- Jak se v Javě vytváří datový záznam - record?
- K čemu je record dobrý?
- Co všechno record vytváří automaticky?
- Co znamená pojem immutable objekt?

## Otázky

{Prg  
7  
V}

- Jak se v Javě používá vnořená třída?
- K čemu jsou vnořené třídy dobré?

## Otázky

{Prg  
7  
V}

- Jak se v Javě používá anonymní třída?
- K čemu se anonymní třídy používají?
- Co je to lambda výraz?
- Jak se v Javě používají lambda výrazy?
- Uveď příklady typického využití lambda výrazů v javovských programech.
- Jaký je vztah mezi lambda výrazem a anonymní třídou?