Objektově orientované programování

Jan Faigl

Katedra počítačů
Fakulta elektrotechnická
České vysoké učení technické v Praze

Přednáška 7

A0B36PR1 – Programování 1

Část 1 – Třídy a objekty

Část 2 – Vztahy mezi objekty

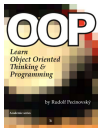
Část 3 – Objektově orientované programování (v Javě)

Část I

Třídy, objekty a objektově orientované programování

Literatura

- Learn Object Oriented Thinking & Programming, *Rudolf Pecinovský* Academic series 2013, ISBN 978-80-904661-9-7



http://pub.bruckner.cz/titles/oop

- Java 7 – Učebnice objektové architektury pro začátečníky, *Rudolf Pecinovský* Grada, 2012

http://knihy.pecinovsky.cz/uoa1_j7/

- Java 8– Úvod do objektové architektury pro mírně pokročile, *Rudolf Pecinovský* Grada, 2014

Datum vydání 17.10.2014

- http://vyuka.pecinovsky.cz

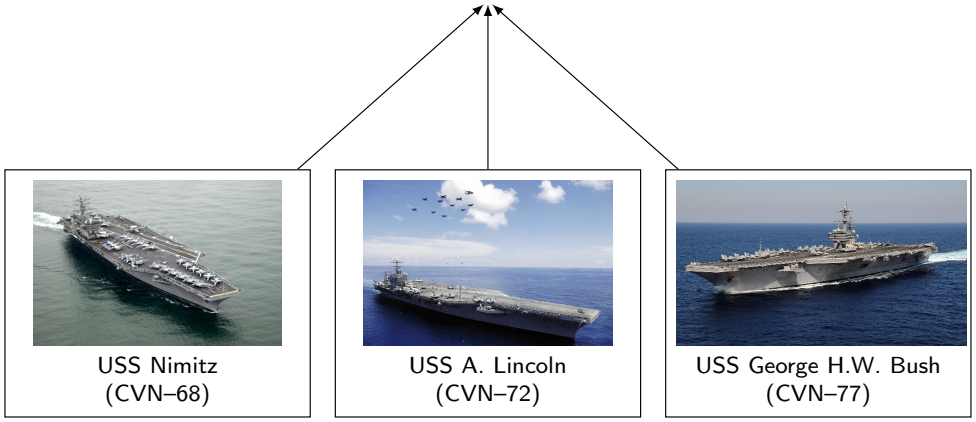
g objektově orientované programování

Třídy a objekty

- Věci okolo nás lze hierarchizovat do tříd (konceptů)
- Každá třída je reprezentována svými prvky (objekty dané třídy)
- Každá třída je charakterizována svými vlastnostmi, funkčními možnostmi a parametry

Příklad – Třídy lodí

Třída lodí Nimitz



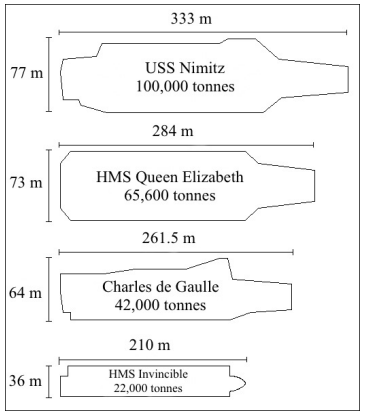
- Třída Nimitz (definice)
  - Metody: řídit loď, zastavit, zadokovat
  - Data (parametry): délka, výtlak, rychlost
- Objekty: jednotlivé lodě odpovídají třídě, ale mají svá specifika

  Posádku, náklad

Třídy a objekty

- Jednotlivé třídy letadlových lodí se liší svou velikostí a výtlakem
- Každá loď je však unikátní, přestože v rámci třídy sdílí řadu parametrů s ostatními loděmi stejné třídy
- Například, každá loď má jinou posádku, která se navíc v průběhu nasazení mění

  Loď je objektem, který se v průběhu svého života mění.



Hierarchie tříd lodí

- Lodě jsou kategorizovány podle svého účelu a velikosti do tříd, například:
  - Třídy letadlových lodí: Forrestal, Enterprise, Nimitz, Kuznetsov, Gerald R. Ford, Queen Elizabeth
  - Třídy bitevních lodí: Freedom, Independence
- Třída je zastoupena jedním plavidlem nebo několika plavidly, například:
  - Nimitz: Nimitz (CVN-68), Dwight D. Eisenhower (CVN-69), Theodore Roosevelt (CVN-71), Abraham Lincoln (CVN-72), George H.W. Bush (CVN-77)
- Třídy představují vzor

  Reprezentovaný vlajkovou lodí
- Jednotlivé lodě představují instance třídy (objekty)

Příklad objektů lodí – AgentC

- Modelování pohybu lodí v boji proti námořnímu pirátství

http://agents.fel.cvut.cz/projects/agentc

Charakteristika objektově orientovaného programování (OOP)

Metodický přístup řešení výpočetních problémů založený na objektovém programování.

- Abstrakce řešeného problému založena na objektovém popisu
- Objekty představují množinu dat a operací
- Objekty mezi sebou komunikují - zasílají zprávy a reagují na události
- Přístup řešení problému vychází z analogie řešení složitých problémů jak by je řešil člověk
- Základním konstruktem jsou objekty a třídy
- Vychází z objektového modelu popisu řešeného problému
- Těsnější vazba mezi analýzou a návrhem

Objektově orientovaná analýza a návrh

- OO analýza se zabývá modelováním, rozborem a specifikací problému.
  - Abstrakce reálného světa
- OO návrh se zabývá řešením problému.
  - Přidává softwarovou abstrakci
- Hranice mezi fází analýzy a návrhem se stírá:
  - Základní konstrukce (třídy a objekty) se používají stejné.
  - Není přesně definováno co patří do fáze analýzy a co do návrhu.
- Cílem objektově orientované analýzy a návrhu (OOAD) je:
  - popis systému reprezentovaný objektovými diagramy (statická struktura),
  - popis dynamiky a chování systému.

Objektově orientované programování

- Základními konstrukčními prvky OOP jsou třídy a objekty

  OOP nejsou jen třídy a objekty!
- Umožňuje abstrakci a zobecnění řešených problému
- Znovu použitelnost implementovaných kódů
- Kontrolu přístup k datům

OOP je přístup jak správně navrhnout strukturu programu tak, aby výsledný program splňoval funkční požadavky a byl dobře udržovatelný.

Třídy a objekty

- Objekty - reprezentují základní entity OO systému za jeho běhu.
  - Mají konkrétní vlastnosti a vykazují chování
  - V každém okamžiku lze popsat jejich stav
  - Objekty se v průběhu běhu programu liší svým vnitřním stavem, který se během vykonávání programu mění
- Třídy - popisují možnou množinou objektů. Předloha pro tvorbu objektů třídy. Mají:
  - Rozhraní - definuje části objektů dané třídy přístupné zvenčí
  - Tělo - implementuje operace rozhraní
  - Instanční proměnné - obsahují stav objektu dané třídy
- *Každý objekt při svém vytvoření dostává privátní kopii instančních proměnných.*
- *Je-li provedena operace, definovaná pro třídu objektů nad daným objektem, dojde ke změně stavu pouze tohoto objektu.*

Třídy a objekty - vlastnosti

- **Zapouzdření** (encapsulation) je množina služeb, které objekt nabízí navenek. Odděluje rozhraní (interface) a jeho implementaci.
- **Stav** je určen daty objektu.
- **Chování** je určeno stavem objektu a jeho službami (metodami).
- **Identita** je odlišení od ostatních objektů (v prog. jazycích pojmenování proměnných reprezentující objekty určité třídy).

Třídy, objekty a programovací jazyky

- Konkretní implementace objektů a tříd se může v prostředí OO programovacího jazyka mírně lišit.
- Typicky se data a operace třídy rozlišují do kategorií:
  - **Public** - data a operace volně přístupné zvenčí.
  - **Protected** - přístupné pouze v rámci dané třídy a podtříd.
  - **Private** - přístupné pouze v rámci dané třídy.
- Konstruktor - operace pro vznik a inicializaci objektu.
  Konstruktory zpravidla slouží k alokaci zdrojů (nastavení parametrů).
- *Destruktor* - operace rušení objektu.
  Zpravidla slouží k uvolnění alokovaných zdrojů.

  V Javě řeší „garbage collector".

Struktura objektu

- Objekt je kombinací dat a funkcí, které pracují nad těmito daty

  Funkce procedurálního programování
- Objekt je tvořen
  - **Datovými strukturami** – atributy
    - Ovlivňují vlastnosti objektu
    - Jsou to proměnné různých datových typů
    - Data jsou zpravidla přístupná pouze v rámci daného objektu a zvnějšku jsou skryta před jinými objekty

      Zapouzdření (encapsulation)
  - **Metodami** – funkce / procedury
    - Určují chování objektu
    - Definují operace nad daty objektu
    - Metody představují služby objektu, proto jsou často veřejné

      Mohou být deklarovány jako privátní

Princip zapouzdření

- „Utajení" vnitřního stavu objektu
- Jiné objekty nemohou měnit stav objektu přímo a způsobit tak chybu

  Např. konzistence hodnot více proměnných
- Metody objektu umožňují objektu komunikovat se svým okolím, tvoří jeho **rozhraní**
- Proměnné (data) objektu nejsou z vnějšku objektu přístupné, pro přístup k nim lze využít pouze metody
- Zapouzdření umožňuje udržovat a spravovat každý objekt nezávisle na jiném objektu. Umožňuje **modularitu** zdrojových kódů.

Komunikace mezi objekty

- V OO systému interagují objekty mezi sebou zasíláním zpráv požadavků na provedení služeb poskytovaných objektem
- Objekty tak mezi sebou komunikují prostřednictvím zpráv, které jsou realizovány (implementovány) metodami
- Pokud jeden objekt požaduje po jiném objektu, aby vykonal nějakou činnost, zašle mu zprávu ve tvaru:
  - Objekt, na kterém se má akce provést
    Referenční proměnná odkazující na objekt, např. String
  - Činnost, která se má vykonat
    Metoda (procedura, funkce), např. `compareTo`
  - Seznam parametrů volané metody
    Parametry funkce
- Zpráva neobsahuje popis jak činnost vykonat, ale pouze co provést
  Konkrétní způsob implementace nemusí být dopředu (v průběhu kompilace) znám (viz např. později diskutované virtuální metody)

Vztahy mezi objekty

- V OO systému interagují objekty mezi sebou prostřednictvím zasílání zpráv (messages) požadavků na provedení služeb poskytovaných objektem
  1. Po obdržení zprávy objekt vyvolá požadovanou metodu
  2. Případně zašle výsledek
- Objekt poskytující službu se často nazývá *server*
- Objekt žádající o službu se nazývá *klient*
- Mezi objekty je relace–asociace, volá-li objekt služby jiného objektu
- Úkolem OOD je explicitně definovat vztahy mezi objekty
  Návrhu – Object Oriented Design (OOD)
- S relacemi mezi objekty souvisí viditelnost a vazby mezi objekty

Relace typu klient/server a vzájemná viditelnost objektů

Základní možnosti vazeb mezi objekty:

- Objekt–server je globální vůči klientovi
- Objekt–server je parametrem některé operace (metody/funkce) klienta, který zasílá zprávu
- Objekt–server je částí objektu klienta
- Objekt–server je lokálně deklarován v rámci operace (metody/funkce)

- Globální server může mít příliš široký definiční obor. Vhodné pouze pokud je objekt široce upotřebitelný
- Server, který je parametrem metody je přístupný pouze nepřímo voláním dané metody
- Server, který je součástí klienta (data-member třídy) zaniká s destrukcí daného objektu daného klienta

Informativní

Příklad třídy jako datového typu – třída `Complex`

- Třída `Complex` – představuje třídu datového typu, jejíž objektový návrh a implementace vychází z konceptu zapouzdření
- Datové položky:
  - Hodnoty typu `double` pro reprezentaci reálné a imaginární části (dvojice čísel)
- Metody: tvoří množinu operací obvyklých pro operace nad komplexními čísly
  - absolutní hodnota, sčítání, odčítání, násobení a dělení

Uvedený příklad je implementací třídy v Javě

Třída Complex 1/6

```
public class Complex {

   //data fields
   private double re = 0.; //data polozka (atribut)
   private double im = 0.; //data polozka (atribut)
   ...
```

- Definice třídy je uvozena klíčovým slovem **class** následovaném jménem třídy
- Kódovací konvence doporučuje psát jméno třídy s prvním písmenem velkým
- Veřejná třída se specifikuje klíčovým slovem (modifikátorem) **public** před `class`
- Datové položky (atributy) se zapisují podobně jako deklarace proměnných

Kódovací konvence doporučuje zapisovat datové položky jako první

Třída Complex 2/6

```
public class Complex {
...
public Complex() {}
public Complex(double r) {
   re = r;
}
public Complex(double r, double i) {
   re = r;
   im = i;
}
```

- Za datovými položkami následují definice **konstruktoru**(ů)
- Konstruktor je metoda stejného jméno jako jméno třídy a nemá návratovou hodnotu
- Konstruktor je volán při vytvoření objektu příkazem **new**, který vrací reference (adresu), kde je objekt uložen v paměti

Třída Complex 3/6

```
public class Complex {
   ...
   //methods (operations)
   public double getAbs() {
      return Math.sqrt(re * re + im * im);
   }

   public Complex plus(Complex b) {
      double r = re + b.re; // r je lokalni promenna
                            // re atribut objektu
      double i = im + b.im;
      return new Complex(r, i);
   }
   ...
```

- Metody jsou funkce s návratovým typem a specifikací přístupových práv

Třída Complex 4/6

```
public class Complex {
   ...
   public Complex minus(Complex b) {
      Complex a = this;
      return new Complex(a.re - b.re, a.im - b.im);
   }

   public Complex times(Complex b) {
      Complex a = this;
      double r = a.re * b.re - a.im * b.im;
      double i = a.re * b.im + a.im * b.re;
      return new Complex(r, i);
   }
```

- Uvnitř metody můžeme použít operátor **this**
- **this** je implicitní odkaz na objekt, na který byla metoda zavolána

Třída Complex 5/6

```
public class Complex {
   ...
   public String toString() {
      if (im == 0) {
         return re + "";
      } else if (re == 0) {
         return im + "i";
      } else if (im <  0) {
         return re + " - " + (-im) + "i";
      }
      return re + " + " + im + "i";
   }
```

- **toString** je metoda každého objektu, která vrací řetězec představující znakovou reprezentaci objektu *„Dědí od třídy `Object`"*
- Pokud není předefinována vrací jméno třídy + hash kód

Překrytí je realizováno dynamickou vazbou (polymorfismu)

Třída Complex 6/6

```
public class Complex {
   ...
   public static Complex plus(Complex a, Complex b) {
      double r = a.re + b.re;
      double i = a.im + b.im;
      Complex sum = new Complex(r, i);
      return sum;
   }
```

Statické metody:

- jsou uvozeny klíčovým slovem **static**
- jsou to metody třídy a nejsou svázány s objektem
- jsou přístupné i bez vytvoření instance třídy (objektu)
- nemají přístup k instančním proměnným (datovým položkám)

Ty se vytvářejí až s vytvořením objektu operátorem `new`

Instance třídy Complex 1/2

```
public static void main(String[] args) {
   Complex c1 = new Complex(2);
   Complex c2 = new Complex(2, 1);

   System.out.println("New complex: " + new Complex());
   System.out.println("Complex var c1: " + c1);
   System.out.println("Complex var c2: " + c2);

   System.out.println("Complex var |c1|: " + c1.getAbs());
   System.out.println("Complex var |c2|: " + c2.getAbs());

   System.out.println("Complex var c1-c2: " + c1.minus(c2));
   System.out.println("Complex var c1+c2: " + c1.plus(c2));
   System.out.println("Complex var c1*c2: " + c1.times(c2));

   System.out.println("Complex: (1 + j) + (1 - j): " +
       Complex.plus(new Complex(1, 1), new Complex(1, -1)));
}
```

- Objekty (instance třídy) `Complex` vytváříme operátorem **new**

Instance třídy Complex 2/2

- Příklad výpisu:

```
java DemoComplex

New complex: 0.0
Complex var c1: 2.0
Complex var c2: 2.0 + 1.0i
Complex var |c1|: 2.0
Complex var |c2|: 2.23606797749979
Complex var c1-c2: -1.0i
Complex var c1+c2: 4.0 + 1.0i
Complex var c1*c2: 4.0 + 2.0i
Complex: (1 + j) + (1 - j): 2.0
```

`lec07/Complex.java` a `DemoComplex.java`

Přístup k datovým položkám

- Datové položky reprezentující reálnou a komplexní část jsou ve třídě `Complex` skryty.

Princip zapouzdření

- Pro přístup k nim, můžeme implementovat metody nazývané

- **getter** – „čtení"

```
public class Complex {
   ...
   public double getRe() {
      return re;
   }
   public double getIm() {
      return im;
   }
   ...
}
```

- **setter** – „zápis"

```
public class Complex {
   ...
   public void setRe(double re) {
      this.re = re;
   }
   public void setIm(double im) {
      this.im = im;
   }
   ...
}
```

Jakou má výhodu přistupovat k proměnným přes metody?

Část II

Vztahy mezi objekty

Agregace

Vztah mezi objekty agregace reprezentuje vztah typu „je tvořeno/je součástí"

Příklad

Je-li objekt ***A*** agregací ***B*** a ***C***, pak objekty ***B*** a ***C*** jsou obecně obsaženy v ***A***

Hlavním důsledkem je fakt, že ***B*** ani ***C*** nemohou přežít bez ***A***

V tomto případě hovoříme o kompozici objektů

Příklad implementace

```
class GraphK { //kompozice
   private Edge[] edges;
}

class GraphA { //agregace
   private Edge[] edges;
   public GraphA(Edge[] edges) {
      this.edges = edges;
   }
}
```

```
class Edge {
   private Node v1;
   private Node v2;
}

class Node {
   private Data data;
}
```


Inheritance - dědičnost

Založení definice a implementace jedné třídy na jiné existující třídě
Třída ***B*** dědí od třídy ***A*** pak:

- Třída ***B*** je podtřídou (subclass) nebo odvozenou třídou (derived class) třídy ***A***
- Třída ***A*** je nadtřídou (superclass) nebo základní třídou (base class) třídy ***B***

Podtřída ***B*** má obecně dvě části:

- Odvozená část je zděděna od ***A***
- Nová inkrementální část (incremental part) obsahující definice a kód přidaný třídou ***B***

Dědičnost (inheritance), pokračování

- Inheritance je také označována jako relace typu is-a
  - Objekt typu ***B*** je také instancí objektu typu ***A***
- Vlastnosti z ***A*** zděděné v ***B*** je možné předefinovat:
  - Změna viditelnosti
  - Jiná implementace operací
- Inheritanční relace vytváří objektové hierarchie
  - Funkce podtříd lze soustředit do jejich nadtříd
  - Lze vytvářet abstraktní třídy, ze kterých je možné další třídy vytvářet specializací

Kategorie dědičnosti

- Striktní dědičnost (strict inheritance) - podtřída přebírá od nadtřídy vše a přidává vlastní metody/atributy. Všechny členy nadtřídy jsou v podtřídě k dispozici. Respektuje přesně zásady „is-a" hierarchií
- Nestriktní dědičnost (nonstrict inheritance) - podtřída odvozuje od nadtřídy pouze některé atributy nebo metody (redefinuje)
- Vícenásobná dědičnost - třída dědí od více nadtříd

V Javě není podporována, řeší se implementací rozhraní

Polymorfismus

- Polymorfismus (mnohotvárnost) se v OOD projevuje tak, že se můžeme stejným způsobem odvolávat na různé objekty
- Pracujeme s objektem, jehož skutečný obsah je dán okolnostmi až za běhu programu
- Polymorfismus objektů - Nechť třída ***B*** je podtřídou třídy ***A***, pak objekt třídy ***B*** můžeme použít všude tam, kde je očekáván objekt třídy ***A***
- Polymorfismus metod - Vyžaduje dynamické vázání, statický a dynamický typ třídy
  - Nechť třída ***B*** je podtřídou třídy ***A*** a redefinuje metodu *m()*
  - Proměnná *x* statického typu ***B***, dynamický typ může být ***A*** nebo ***B***
  - Jaká metoda se skutečně volá pro *x.m()* závisí na dynamickém typu

Dědičnost, polymorfismus a virtuální metody

- Vytvoření dynamické vazby je zpravidla v OO programovacím jazyce realizováno virtuální metodou
- Redefinované metody, které jsou označené jako virtuální, mají dynamickou vazbu na konkrétní dynamický typ

Polymorfismus příklad

```
class A {
  void info() {
    System.out.println("A");
  }
};
```

```
class B extends A {
  void info() {
    System.out.println("B");
  }
};
```

```
A a = new A();
B b = new B();

a.info(); // volani metody info tridy A
b.info(); // volani metody info tridy B
a = b;
a.info(); // dynamicka vazba volani metody tridy B
```

Výstup:
A
B
B

lec07/DemoPolymorphism.java

Část III

Objektově orientované programování (v Javě)

Objektový přístup programování

- Modelování problému jako systému spolupracujících tříd
- Třída modeluje jeden koncept
- Třídy umožňují generování instancí, objektů příslušné třídy
- Jednotlivé objekty spolu spolupracují

  Zasíláním si zpráv
- Třída je „vzorem" pro strukturu a vlastnosti generovaných objektů
- Každý objekt je charakteristický specifickými hodnotami svých atributů a společnými vlastnostmi třídy

Třídy a objekty

- **Třída** – šablona pro generování konkrétních instancí třídy, tj. **objektů**, je tvořena členy třídy (datové položky a metody)
  - data, atributy – určují stav objektů
  - funkce, metody – určují schopnosti objektů
- **Objekt** – instance třídy
  - Jednotlivé instance třídy (objekty) mají stejné metody, ale nacházejí se v různých stavech

    Stav objektu je určen hodnotami instančních proměnných
  - Schopnosti objektu jsou dány instančními metodami
- V jazyku Java lze objekty (instance tříd) vytvářet pouze dynamicky operátorem **new**

  Objekty jsou alokovány na haldě (heap).
- Přistupovat k nim lze prostřednictvím referenčních proměnných

  Podobně jako u pole.

Datové položky třídy a instance

- Datové položky třídy
  - Jsou společné všem instancím vytvořeným z jedné třídy
  - Nejsou vázaný na konkrétní instanci
  - Jsou společné všem instancím třídy
  - V Javě jsou uvozeny klíčovým slovem **static**
- Datové položky instance
  - Tvoří vlastní sadu datových položek objektu
  - Jsou to tzv. proměnné instance
  - Jsou iniciovány při vytvoření instance

    V konstruktoru při vytvoření instance voláním new
  - Existují po celou dobu života instance
  - Proměnné jedné instance jsou nezávislé na proměnných instance jiné

Metody třídy a instance

- Metody třídy
  - Nejsou volány pro konkrétní instance
  - Představují zprávu zaslanou třídy jako celku
  - Mohou pracovat pouze s proměnnými třídy

    Nikoliv s proměnnými instance
  - V Javě jsou uvozeny klíčovým slovem **static**
  - Jsou to tzv. statické metody
- Metody instance
  - Jsou volány vždy pro konkrétní instanci třídy
  - Představují zprávu zaslanou konkrétní instanci
  - Pracují s proměnnými instance i s proměnnými třídy
  - Lze volat pouze až po vytvoření konkrétní instance

Přístup ke členům třídy

- Podle principu zapouzdření jsou některé členy třídy označována jako soukromé (privátní) a jiné jako veřejné.
- Programátor předepisuje k jakým položkám lze přistupovat a modifikovat je
- Přístup ke členům třídy je určen modifikátorem přístupu
  - **public:** – přístup z libovolné třídy
  - **private:** – přístup pouze ze třídy, ve které byly deklarovány
  - **protected:** – přístup ze třídy a z odvozených tříd
  - Bez uvedení modifikátoru je přístup povolen v rámci stejného balíčku package

Řízení přístup ke členům třídy

Modifikátor	Přístup			
	Třída	Balíček	Odvozená třída	„Svět"
public	✓	✓	✓	✓
protected	✓	✓	✓	✗
bez modifikátoru	✓	✓	✗	✗
private	✓	✗	✗	✗

http://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html

Konstruktor třídy

- Představuje speciální metodu
- Jméno metody je shodné se jménem třídy

Jediná metoda začínající velkým písmenem.

- Vytvoří objekt
- Nastaví vlastnosti objektu
- Neobsahuje návratový typ – nic nevrací, vytváří objekt
- Může být přetížen pro různé typy a počty parametrů
- Není-li konstruktor předepsán, je vygenerován konstruktor s prázdným seznamem parametrů
  - Je-li konstruktor deklarován, implicitní zaniká

Přetěžování konstruktorů

- Příklad konstruktoru pro vytvoření instance komplexního čísla
- Při vytváření specifikujeme pouze reálnou nebo reálnou i imaginární část

```
public class Complex {
   ...
   public Complex(double r) {
      re = r;
   }

   public Complex(double r, double i) {
      re = r;
      im = i;
   }
   ...
}
```

V kódu obou konstruktorů je duplicitní kód, kterému se snažíme vyhnout (jednodušší opravy)!

Vzájemné volání konstruktorů 1/2

- V konstruktoru můžeme volat jiný konstruktor použitím operátoru **this**

```
public class Complex {
   ...
   public Complex(double r) {
      re = r;
   }

   public Complex(double r, double i) {
      this(r); //volani konstruktoru Complex(double r)
      im = i;
   }
   ...
}
```


Vzájemné volání konstruktorů 2/2

- Můžeme vytvořit jeden „obecný" konstruktor, který bude volán z ostatních konstruktorů

```
public class Complex {
   ...
   public Complex(double r, double i) {
      re = r;
      im = i;
   }

   public Complex(double r) {
      this(r, 0.0);
   }

   public Complex() {
      this(0.0, 0.0);
   }

   ...
}
```

Shrnutí vlastností konstruktorů

- Jméno konstruktoru je identické se jménem třídy
- Konstruktor nemá návratovou hodnotu

Ani `void`

- Předčasně lze ukončit činnost konstruktoru voláním **return**
- Konstruktor má parametrovou část jako metoda – může mít libovolný počet a typ parametrů
- V těle konstruktoru můžeme použít operátor **this** jako odkaz na příslušný konstruktor s počtem, pořadím a typem parametrů

Nepíšeme jméno třídy

- Konstruktor je zpravidla vždy **public**
- Privátní (**private**) konstruktor použijeme například pro:
  - Třídy obsahující pouze statické metody (utility)

Zakážeme tak vytváření instancí.

  - Třídy obsahující pouze konstanty
  - Takzvané singletony (singletons)

např. „továrny na objekty"

Příklad třídy jako rozšířeného datového typu

- Využijeme zapouzdření a implementujeme třídu reprezentující dvou rozměrnou matici hodnot typu `double`

```
public class Matrix {

   private final double[][] values;
   private final int rows;
   private final int cols;

   ...
}
```

- V konstruktoru vytváříme pole polí hodnot double a nastavíme proměnnou `double[][] values`
- Položka `values` je privátní a uživateli je skryta
- Rozměr matice je fixní po dobu života objektu (matice)

Příklad – Třída `Matrix` – Konstruktor

- V konstruktoru testujeme přípustnost rozměru matice

```
pulic class Matrix {
   ...
   public Matrix(int rows, int cols) {
      if (rows <=0 || cols <= 0) {
         throw new IllegalArgumentException();
      }
      this.rows = rows;
      this.cols = cols;
      values = new double[rows][];
      for (int i = 0; i < rows; i++) {
         values[i] = new double[cols];
      }
   }
   ...
}
```


Příklad – Třída `Matrix` – Přístupové metody

- Přístup na datové položky implementujeme tzv. „accessory"
- Pro čtení použijeme „gettery"

```
public class Matrix {
   ...
   public int getNumberOfRows() {
      return rows;
   }

   public int getNumberOfCols() {
      return cols;
   }
   ...
}
```

Příklad – Třída Matrix – Přístupová metoda buňky matice

- Při přístupu na buňku matice testujeme přípustnost indexů

```
public class Matrix {
   ...
   public double getValueAt(int r, int c) throws
    IllegalAccessException {
      if (r < 0 || r >= rows || c < 0 || c >= cols) {
         throw new IllegalAccessException();
      }
      return values[r][c];
   }
   ...
}
```


Příklad – Třída Matrix – Nastavení hodnoty matice

- Podobně testujeme nastavení hodnoty buňky

```
public class Matrix {
   ...
   public void setValueAt(int r, int c, double v)
    throws IllegalAccessException {
      if (r < 0 || r >= rows || c < 0 || c >= cols) {
         throw new IllegalAccessException();
      }
      values[r][c] = v;
   }
   ...
}
```

Případně můžeme implementovat společnou metodu pro testování indexů.

Příklad – Třída Matrix – Přetížený konstruktor

- Výhodou zapouzdření je, že nemusíme kontrolovat, zda-li jsou dílčí pole (sloupce) alokovány
- Alokaci garantuje konstruktor

Např. v případě nedostatku paměti, selže volání konstruktoru.

- Pro vytvoření matice stejných rozměrů můžeme využít přetížený konstruktor a operátor **this**

```
public class Matrix {
   ...
    public Matrix(Matrix m) {
        this(m.rows, m.cols);
    }
   ...
}
```

Pro vytváření kopií objektů můžeme také implementovat rozhraní `Clonable` předepisující metodu `clone` a dále pak vytvářet mělké či hluboké kopie.

Příklad – Třída Matrix – Vyplnění matice

- Podobně jako v příkladu ze 4. přednášky můžeme implementovat vyplnění matice náhodnými hodnotami (např. pro testovací účely)

lec04/DemoArrayOfArray.java

```
public class Matrix {
   ...
   public void fillRandom() {
      for (int r = 0; r < rows; ++r) {
         for (int c = 0; c < cols; ++c) {
            values[r][c] = (int) (Math.random() * 10);
         }
      }
   }
   ...
}
```

Příklad – Třída Matrix – Tisk matice

- Předefinováním metody toString můžeme využít pro tisk metody System.print

```
public class Matrix {
   public @Override String toString() {
      StringBuilder str = new StringBuilder();
      for (int r = 0; r < rows; ++r) {
         for (int c = 0; c < cols; ++c) {
            str.append(
                  String.format("%s%4.1f",
                     c > 0 ? " " : "", values[r][c]));
         }
         str.append("\n");
      }
      return str.toString();
   }
```

@Override – je anotace indikující úmysl přepsat metodu předka

Příklad – Třída Matrix – Součet

- Podobně můžeme implementovat metodu pro součet dvou matic aniž bychom museli explicitně kontrolovat každý řádek matic

Srovnejte s implementací ze 4. přednášky

```
public class Matrix {
   public Matrix sum(Matrix a) {
      if (!(rows == a.rows && cols == a.cols)) {
         return null;
      }
      Matrix ret = new Matrix(this);
      double[][] m = ret.values;
      for (int r = 0; r < rows; ++r) {
         for (int c = 0; c < cols; ++c) {
            m[r][c] = values[r][c] + a.values[r][c];
         }
      }
      return ret;
   }
```


Příklad – Třída Matrix – Příklad součtu

- Vytvoříme dvě matice, které náhodně vyplníme a sečteme
- Výsledek uložíme do referenční proměnné sum

```
Matrix m1 = new Matrix(3, 3);
Matrix m2 = new Matrix(3, 3);

m1.fillRandom();
m2.fillRandom();
Matrix sum = m1.sum(m2);

System.out.println("M1:\n" + m1);
System.out.println("M2:\n" + m2);
System.out.println("sum:\n" + sum);
```

lec07/Matrix.java, lec07/DemoMatrix.java

Příklad – Třída Matrix – Součin

- Podobně jako součet implementujeme součin

Pro jednoduchost předpokládáme čtvercové matice

```
public Matrix product(Matrix a) {
   final int n = rows;
   if (!(cols == rows && a.rows == n && a.cols == n)) {
      return null;
   }
   Matrix ret = new Matrix(this);
   for (int i = 0; i < n; ++i) {
      for (int j = 0; j < n; ++j) {
         ret.values[i][j] = 0.0;
         for (int k = 0; k < n; ++k) {
           ret.values[i][j] += values[i][k] * a.values[k][j];
   } } }
   return ret;
}
```

final int n vs final int N

Příklad – Třída Matrix – Součin (jinak)

- Součin můžeme také implementovat alternativně s využitím transpozice

```
public Matrix productTrans(Matrix a) {
   final int n = rows;
   Matrix mTmp = new Matrix(this);
   for (int r = 0; r < n; ++r) { //transpozice matice a
      mTmp.values[r][r] = a.values[r][r];
      for (int c = r + 1; c < n; ++c) {
         mTmp.values[r][c] = a.values[c][r];
         mTmp.values[c][r] = a.values[r][c];
      }
   }
   Matrix ret = new Matrix(this);
   for (int i = 0; i < n; ++i) {
      for (int j = 0; j < n; ++j) {
         ret.values[i][j] = 0.0;
         for (int k = 0; k < n; ++k) {
            ret.values[i][j] += m1[i][k] * mTmp.values[j][k];
         }
      }
   }
   return ret;
}
```

Informativní

Příklad – Třída Matrix – product vs productTrans

```
final int N = 1000;
m1 = new Matrix(N, N);
m2 = new Matrix(N, N);
m1.fillRandom();
m2.fillRandom();

long t1 = System.currentTimeMillis();
m1.product(m2);
long t2 = System.currentTimeMillis();
long dt1 = t2 - t1;
System.out.printf("Time %14s: %6d ms%n", "product", dt1);
m1.productTrans(m2);
long t3 = System.currentTimeMillis();
long dt2 = t3 - t2;
System.out.printf("Time %14s: %6d ms%n", "productTrans", dt2);
```

lec07/Matrix.java, lec07/DemoMatrix.java

Program si vyzkoušejte a vysvětlete rozdíl.

Shrnutí přednášky

Diskutovaná témata

- Třídy a objekty
  - Úvod do objektově orientovaného modelování (analýzy a návrhu)
  - Objektově orientované programování (OOP)
  - Struktura objetu a zapouzdření
  - Příklad – Třída Complex
- Vztahy mezi objekty – agregace, dědičnost, polymorfismus
- OOP v Javě
  - Metody a datové položky třídy a instance
  - Řízení přístupu k položkám
  - Konstruktor třídy
  - Příklad – Třída Matrix
- Příště: Dědičnost