Objektově orientované programování

Jan Faigl

Katedra počítačů
Fakulta elektrotechnická
České vysoké učení technické v Praze

Přednáška 8

A0B36PR1 – Programování 1

Část 1 – Třídy a objekty

Třídy a objekty – shrnutí 7. přednášky

Význam metody `main`

Objekty základních typů

Neměnitelné objekty (Immutable objects)

Dědičnost

Příklad

Hierarchie tříd

Část 2 – Příklad tříd geometrických objektů a jejich vizualizace

Zadání

Popis výchozích rozhraní a tříd

Návrh řešení

Implementace

Příklad použití

Část I

Třídy a objekty

Objektově orientované programování (OOP)

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

- **Abstrakce** – koncepty (šablony) organizujeme do tříd, objekty jsou pak instance tříd.
- **Zapouzdření** (encapsulation)
  - Objekty mají svůj stav skrytý, poskytují svému okolí **rozhraní**, komunikace s ostatními objekty zasíláním zpráv (volání metod)
- **Dědičnost** (inheritance)
  - Hierarchie tříd (konceptů) se společnými (obecnými) vlastnostmi, které se dále specializují
- **Polymorfismus** (mnohotvárnost)
  - Objekt se stejným rozhraním může zastoupit jiný objekt téhož rozhraní.

Třída

Popisuje množinu objektu – je jejich vzorem (předlohou) a definuje:

- Rozhraní – části, které jsou přístupné zvenčí

  public, **protected**, **private**, *package*

- Tělo – implementace operací rozhraní (metod), které určují schopnosti objektů dané třídy

  instanční vs statické (třídní) metody

- Datové položky – atributy základních i složitějších datových typů a struktur

  kompozice objektů

  - Instanční proměnné – určují stav objektu dané třídy
  - Třídní (statické) proměnné – společné všem instancím dané třídy

Struktura objektu

- Hodnota objektu je strukturovaná, tj. skládá se z dílčích hodnot, které mohou být obecně různého typu

  Heterogenní datová struktura – na rozdíl od pole

- Objekt je abstrakcí paměťového místa skládajícího se z částí, ve kterých jsou uloženy dílčí hodnoty – **položky objektu**
  - atributy, instanční proměnné
- Datové položky jsou označeny jmény a mohou být třídou zveřejněny

  Dle principu zapouzdření se však zpravidla nezveřejňují.

Objekt:

- Instance třídy – lze vytvářet pouze dynamicky operátorem **new**

  v Javě

- **Referenční proměnná**

  Hodnota proměnné „odkazuje" na místo v paměti, kde je objekt uložen

- K atributům a metodám se přistupuje prostřednictvím **.**

Statické datové položky a metody

- Statické datové položky „patří" třídě
  - Jsou součástí třídy, existují i bez (objektu) instance třídy
  - Vznikají při startu programu („nahrání" třídy)
- Statické metody jsou metody třídy
  - Mají přístup ke statickým položkám třídy
- Důsledek:
  - Metody můžeme volat i bez vytvoření objektu
  - Představují *procedury* a *funkce*

Příklad

- Datové položky třídy **Math**, např. **Math.E**, **Math.PI** nebo matematické funkce **Math.sin()**, **Math.sqrt()**
- Konverzní funkce třídy **String**, např. **String.valueOf**() nebo konverzní metody třídy **Integer** pro parsování řetězce **Integer.parseInt(String s)**

Vytvoření objektu – Konstruktor třídy

- Instance třídy (objekt) vzniká voláním operátoru **new** s argumentem jména třídy
- Při vzniku je volána speciální metoda třídy zvána **konstruktor**, ve které můžeme nastavit hodnoty instančních proměnných
- Konstruktor nemá návratový typ, jmenuje se stejně jako třída a můžeme jej přetížit pro různé typy a počty parametrů
- V konstruktoru můžeme volat jiný konstruktor prostřednictvím operátoru **this**

*Další speciální operátor je **super***

- Zpravidla je **public**, ale může být i **private**

Private používáme např. pro „knihovny" statickým metod

Vztahy mezi objekty

- Objekty mohou obsahovat jiné objekty
- Agregace / kompozice
- Definice třídy může být založena na definici již existující třídy vzniká tak vztah mezi třídami
  - Základní třída (nadtřída/super class) a odvozená třída (derived class)
  - Dědičný vztah se přenáší i na objekty jako instance příslušných tříd

    Důsledek je, že můžeme objekty přetypovat na instance třídy předka.
- Objekty mezi sebou komunikují prostřednictvím metod, které mají vzájemně přístupné

Datové položky objektů

- Dle principu zapouzdření jsou datové položky zpravidla **private**
- Přístup k položkám je přes metody, tzv. „accessory", které vrací/-nastavují hodnotu příslušné proměnné („getter" a „setter")

```
public class DemoGetterSetter {
   private int x;
}
```

- „Accessory" lze vytvořit mechanicky a vývojová prostředí zpravidla nabízí automatické vygenerování jejich zdrojového kódu

```
public class DemoGetterSetter {
   private int x;

   public int getX() {
      return x;
   }

   public void setX(int x) {
      this.x = x;
   }
}
```

Viz Alt+Insert v Netbeans

Statická metoda **main**

- Deklarace hlavní funkce

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

  představuje „spouštěč" programu
- Musí být statická, je volána dříve než se vytvoří objekt
- Třída nemusí obsahovat funkci **main**
  - Taková třída zavádí prostředky, které lze využít v jiných třídách
  - Jedná se tak o „knihovnu" funkcí a procedur nebo datových položek (konstant)
- Kromě spuštění programu může funkce **main** obsahovat například testování funkčnosti objektu nebo ukázku použití metod objektu

Např. jak je použito v příkladech na přednáškách a cvičení

*Třída s hlavní funkcí **main** tvořící základ programu je specialitou jazyka Java. V jiných jazycích např. C/C++, lze program vytvořit bez použití třídy.*

Objekty pro základní typy

- Každý primitivní typ má v Javě také svoji třídu:
  - **Char**, **Boolean**
  - **Byte**, **Short**, **Integer**, **Long**
  - **Float**, **Double**
- Třídy obsahují metody pro převod čísel a metody pro parsování čísla z textového řetězce
  - např. **Integer.parseInt(String s)**
- Dále také rozsah číselného typu minimální a maximální hodnoty
  - např. **Integer.MAX_VALUE**, **Integer.MIN_VALUE**

http://docs.oracle.com/javase/8/docs/api/java/lang/Number.html

Referenční proměnné objektů tříd primitivních typů

- Referenční proměnné objektů pro primitivní typy můžeme používat podobně jako základní typy

```
Integer a = 10;
Integer b = 20;

int r1 = a + b;
Integer r2 = a + b;
System.out.println("r1: " + r1 + " r2: " + r2);
```

- Stále to jsou však referenční proměnné (odkazující na adresu v paměti)
- Obsah objektu nemůžeme měnit, jedná se o tzv. **immutable** objekty
- Příklad volání funkce / metody

Porovnejte s voláním a předáváním hodnoty prostřednictvím třídy DoubleValue ze 7. cvičení

lec08/DemoObjectsOfBasicTypes.java

Neměnitelné objekty (Immutable objects)

- Objekty, které v průběhu života nemění svůj stav
- Například instance třídy **String**
- Modifikace objektu není možná a je nutné vytvořit objekt nový
- Mají výhodu v případě paralelního běhu více výpočetních toků
- Vytváření nových objektů je zpravidla spojeno s režií

Ta však může být zanedbatelná

http://docs.oracle.com/javase/tutorial/essential/concurrency/immutable.html

- Definice neměnitelného objektu
  - Všechny datové položky jsou **final** a **private**

Reference na neměnitelné objekty!

  - Neimplementujeme „settery" pro modifikaci položek
  - Zákaz přepisu metod v potomcích (**final** modifikátor u metod)

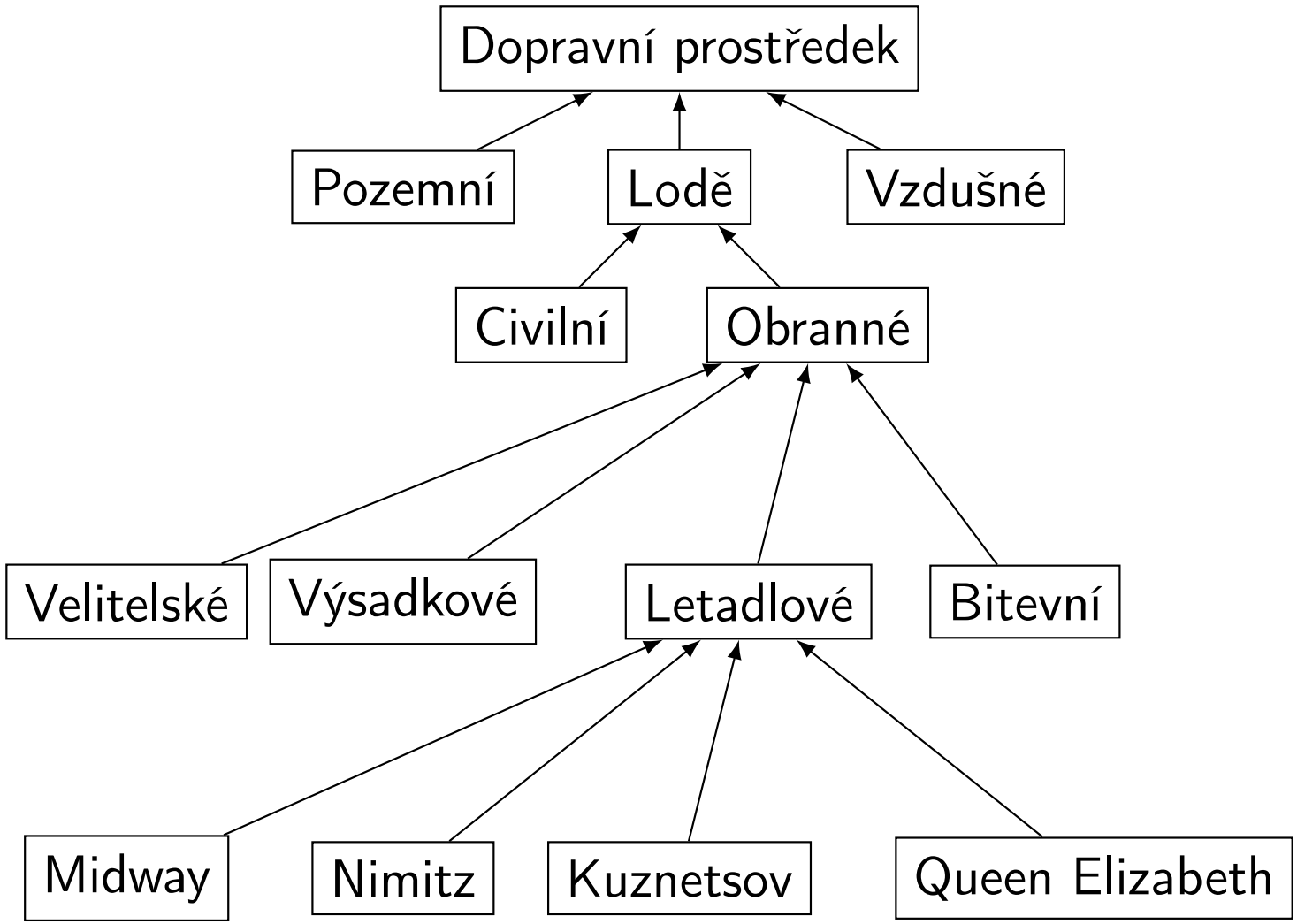
http://docs.oracle.com/javase/tutorial/essential/concurrency/imstrat.html

Základní vlastnosti dědičnosti

- Dědičnost je mechanismus umožňující
  - Rozšiřovat datové položky tříd nebo je také modifikovat
  - Rozšiřovat nebo modifikovat metody tříd
- Dědičnost umožňuje
  - Vytvářet hierarchie tříd
  - „Předávat" datové položky a metody k rozšíření a úpravě
  - Specializovat („upřesňovat") třídy
- Mezi hlavní výhody dědění patří:
  - Zásadním způsobem přispívá ke znovupoužitelnosti programového kódu

    Spolu s principem zapouzdření.
  - Dědičnost je základem polymorfismu

Příklad hierarchie tříd



Odvozené třídy, polymorfismus a praktické důsledky

- Odvozená třída dědí metody a položky nadtřídy, ale také může přidávat položky nové
  - Můžeme rozšiřovat a specializovat schopnosti třídy
  - Můžeme modifikovat implementaci metod
- Objekt odvozené třídy může „vystupovat" místo objektu nadtřídy
  - Můžeme například využít efektivnější implementace aniž bychom modifikovali celý program.
  - Příklad různé implementace maticového násobení

viz `lec07/Matrix.java`, `lec07/DemoMatrix.java`

Třída zapouzdřující maticové operace

- Naším úkolem je provést sadu maticových výpočtů, např.:

```
public Matrix compute(int n, Matrix matrix) {
    Matrix m1 = matrix.createMatrix(n, n);
    Matrix m2 = matrix.createMatrix(n, n);
    m1.fillRandom();
    m2.fillRandom();

    Matrix semiResult1 = m1.sum(m2);
    Matrix semiResult2 = m1.difference(m2);
    return semiResult1.product(semiResult2);
}
```

viz `lec08/**/Matrix.java`, `lec08/**/DemoMatrix.java`

- Pro začátek implementuje přímočaré násobení matice

Už však tušíme, že se to dá také udělat jinak, proto navrhneme třídu Matrix „rozšiřitelnou".

Třída reprezentující matici

- Volání metody konkrétní třídy závisí jakého typu je referenční proměnná (objekt)

  Jaké jméno třídy použijeme při volání operátoru ***new***

- Proto ve třídě `Matrix` vytvoříme metodu pro vytvoření instance konkrétní třídy, kterou bude možné v odvozených třídách předefinovat a vytvářet tak instance odvozených tříd

  Vystupovat však budou tyto instance „rozhráním" třídy Matrix

```
class Matrix {
    ...
    protected Matrix createMatrix(int rows, int cols) {
        return new Matrix(rows, cols);
    }
    ...
}
```

- Tuto metodu pak v odvozených třídách modifikujeme, aby vytvářela instance právě definované odvozené třídy.

  To teď ještě udělat nemůžeme, protože odvozené třídy ještě neexistují.

Třída Matrix – operace součtu

- Při implementaci operací pak důsledně používáme pro vytvoření nových objektů (matic) metodu createMatrix

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

Třída Matrix – operace násobení

- Podobně také v implementaci operace násobení

```
public Matrix product(Matrix a) {
    Matrix ret = createMatrix(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;
}
```

Vytváření matice metodou `createMatrix` *místo volání* `new Matrix` *je důležité, aby v odvozených třídách nabízející nové implementace metod vytvářely instance právě těchto odvozených tříd.*

Příklad použití třídy Matrix

- Při výpočtu pak využíváme předaného parametru pro vytvoření matice pro mezi výsledky

```
public Matrix compute(int n, Matrix matrix) {
    Matrix m1 = matrix.createMatrix(n, n);
    Matrix m2 = matrix.createMatrix(n, n);
    m1.fillRandom();
    m2.fillRandom();

    Matrix semiResult1 = m1.sum(m2);
    Matrix semiResult2 = m1.difference(m2);
    return semiResult1.product(semiResult2);
}

Matrix matrix = new Matrix(1, 1);
Matrix results = compute(1000, matrix);
```

Odvozená třída MatrixExtended

- Pokud nejseme spokojeni s rychlostí násobení odvodíme novou třídu MatrixExtended od třídy Matrix
- Přepíšeme pouze metody createMatrix a product
- V konstruktoru zajistíme volání konstruktoru předka přes **super**

```
public class MatrixExtended extends Matrix {

    public MatrixExtended(int rows, int cols) {
        super(rows, cols);
    }

    @Override
    protected Matrix createMatrix(int rows, int cols) {
        return new MatrixExtended(rows, cols);
    }

    @Override
    public Matrix product(Matrix a) {
        ...
    }
}
```

lec08/**/MatrixExtended.java

Výpočet s Matrix nebo MatrixExtended

```
public void start(String[] args) {
    final int N = 1000;
    final boolean FAST_MATRIX = true;

    Matrix matrix = FAST_MATRIX ?
        new MatrixExtended(1, 1) : new Matrix(1, 1);

    long t1 = System.currentTimeMillis();
    Matrix results = compute(1000, matrix);
    long t2 = System.currentTimeMillis();
    System.out.printf("Time is %6d ms%n", (t2 - t1));
}
```

lec08/**/DemoMatrix.java

- Do kódu funkce compute již nezasahujeme

Uvedený příklad slouží k demonstraci jakým způsobem lze využít odvození třídy a její specializace. Zároveň také demonstruje polymorfismus, neboť ve výpočtu funkce `compute` *přistupujeme k maticím prostřednictvím rozhraní třídy* `Matrix` *avšak vlastní objekty mohou být buď instance třídy* `Matrix` *tak odvozené třídy* `MatrixExtended`.

Hierarchie tříd

- V uvedeném příkladu je třída `MatrixExtended` podtřídou třídy `Matrix`
- Podtřída dědí vlastnosti nadtřídy a rozšiřuje třídu o nové vlastnosti
- Zděděné vlastnosti mohou být v podtřídě modifikovány

- Pro instanční metody to znamená:
  - Každá metoda třídy `Matrix` je i metodou třídy `MatrixExtended`

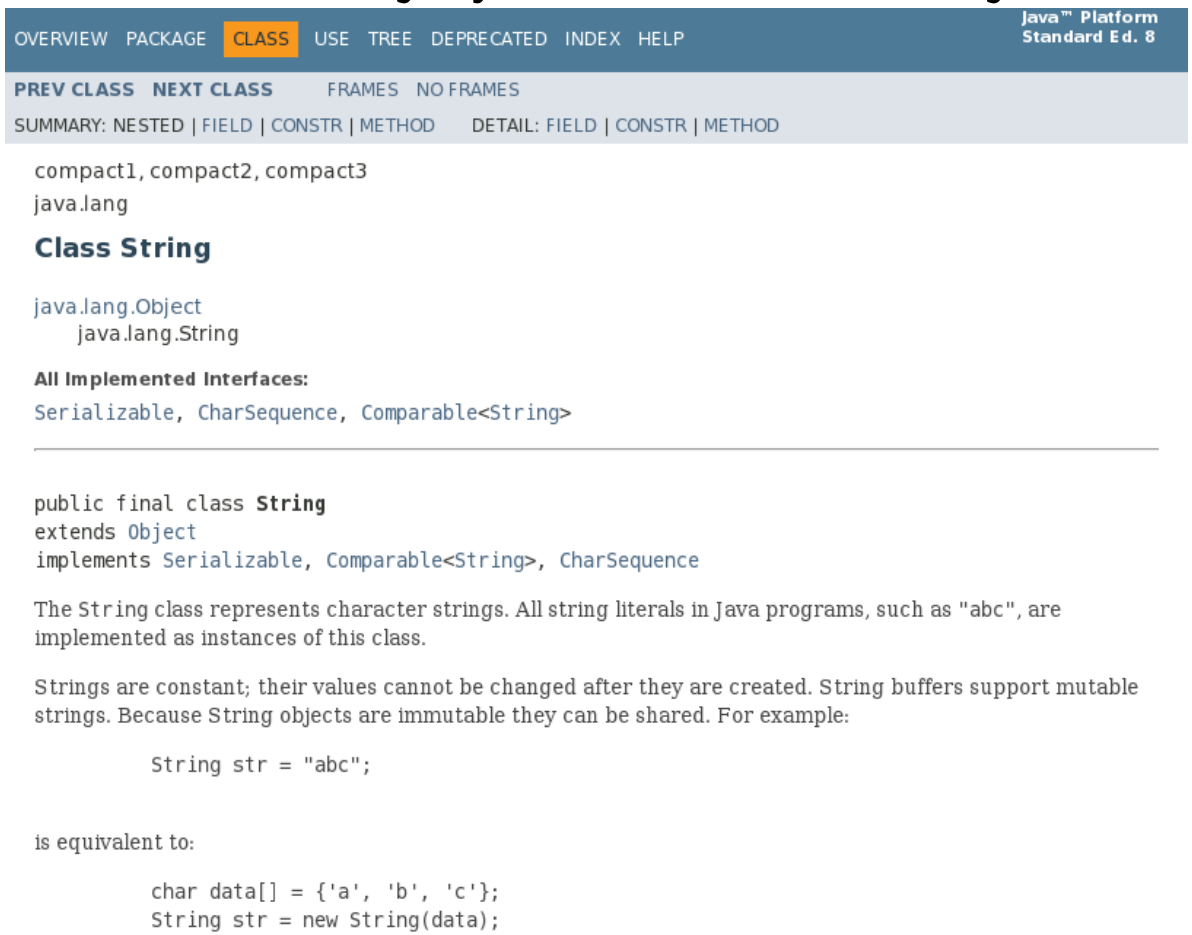
    *V podtřídě však může mít jinou implementaci (**@Override**)*
  - V podtřídě mohou být definovány nové metody

- Pro strukturu objektu to znamená
  - Instance třídy `MatrixExtended` mají všechny členy třídy `Matrix` a případně další části

    *Některé však mohou být nepřístupné (**private**)*

Hierarchie tříd v knihovně JDK

- V dokumentaci jazyka Java můžeme najít následující obrázek



OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP — Java™ Platform Standard Ed. 8

PREV CLASS NEXT CLASS FRAMES NO FRAMES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

compact1, compact2, compact3

java.lang

Class String

java.lang.Object
    java.lang.String

All Implemented Interfaces:

Serializable, CharSequence, Comparable<String>

```
public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence
```

The String class represents character strings. All string literals in Java programs, such as "abc", are implemented as instances of this class.

Strings are constant; their values cannot be changed after they are created. String buffers support mutable strings. Because String objects are immutable they can be shared. For example:

```
String str = "abc";
```

is equivalent to:

```
char data[] = {'a', 'b', 'c'};
String str = new String(data);
```

https://docs.oracle.com/javase/8/docs/api/index.html?java/lang/String.html

Třída String

- Třída `String` je odvozena od třídy `Object`
- Třída implementuje rozhraní `Serializable, CharSequence` a `Comparable<String>`
- Třída je **final** – tj. nemůže být od ní odvozena jiná třída

```
public final class String extends Object {
```

- Třída je **Immutable** – její datové položky nelze měnit

Třída Object

- Třída `Object` tvoří počátek hierarchie tříd v Javě
- Tvoří nadtřídu pro všechny třídy
- Každá třída je podtřídou (je odvozena od) `Object`
  - **class A {}** je ekvivalentní s **class A extends Object {}**
- Třída Object implementuje několik základních metod:
  - **protected Object clone();**

    Vytváří kopii objektu
  - **public boolean equals(Object o);**
  - **Class<?> getClass();**
  - **int hashCode();**
  - **public String toString(o);**

    Vrací textovou reprezentaci objektu
- Také implementuje metody pro synchronizací vícevláknových programů: **wait**, **notify**, **notifyAll**

  Každý objekt je také tzv. „monitorem".

https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html

Metoda toString()

- Metodou je zavedena implicitní typová konverze z typu objektu na řetězec reprezentující konkrétní objekt, např. pro tisk objektu metodou print

Lze využít automatické vytvoření v Netbeans

- Například metoda **toString** ve třídách Complex a Matrix

```
public class Complex {
    ...
    @Override
    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";
    }
}
```

lec07/Complex.java, lec08/**/Matrix.java

Metoda equals()

- Standardní chování

  neporovnává obsah datových položek objektu, ale reference (adresy)

```
public boolean equals(Object obj) {
   return this == obj;
}
```

- Při zastínění můžeme porovnávat obsah datových položek, např.

```
public class Complex {
   @Override
   public boolean equals(Object o) {
      if (! (o instanceof Complex)) {
         return false;
      }
      return re == o.re && im == o.im;
} }
```

- Pro zjištění, zdali je referenční proměnná instancí konkrétní třídy můžeme použít operátor **instanceof**

Metody equals() a hashCode()

- Pokud třída modifikuje metodu `equals()` je vhodné také modifikovat metodu `hashCode()`
- Metoda `hashCode()` vrací celé číslo reprezentující objekt, které je například použito v implementaci datové struktury `HashMap`
- Pokud metoda `equals()` vrací pro dva objekty hodnotu `true` tak i metoda `hashCode()` by měla vracet stejnou hodnotu pro oba objekty
- Není nutné, aby dva objekty, které nejsou totožné z hlediska volání `equals`, měly nutně také rozdílnou návratovou hodnotu metody `hashCode()`

Zlepší to však efektivitu při použití tabulek HashMap, viz dokumentace nebo PR2.

Operátor **super**

- Pokud je potřeba zavolat v podtřídě metodu nebo konstruktor z nadtřídy, je možné využít operátor **super**
- V konstruktoru lze volat buď **super** nebo jiný konstruktor **this**

  Řešíme například tak, že v obecném konstruktoru se všemi parametry voláme `super` *a ve specializovaných konstruktorech voláme obecný konstruktor operátorem* `this`.
- Příklad:
  - Třída `Appender`, která je dále rozšířena ve třídě `AppenderExtended`

`lec08/DemoSuper.java`

Příklad – Appender

```
class Appender {

   protected String str;

   public Appender(String s) {
      this.str = s + "\nConstructor of class A";
   }
   public void append(String s) {
      str = str + "\n Append in class A '" + s + "'";
   }

   @Override
   public String toString() {
      return str;
   }
}
```

Příklad – AppenderExtended

```
class AppenderExtended extends Appender {

    public AppenderExtended(String s) {
        super(s); // call constructor of the super class
    }

    @Override
    public void append(String s) {
        str = str + "\n Append in class B '" + s + "'";
        super.append(s); // call super class method append
    }
}
```

Příklad volání konstruktorů a metody třídy AppenderExtended

- Vytvoření instance třídy AppenderExtended

```
AppenderExtended a = new AppenderExtended("This is B
    object");
a.append("Text");
System.out.println(a);
```

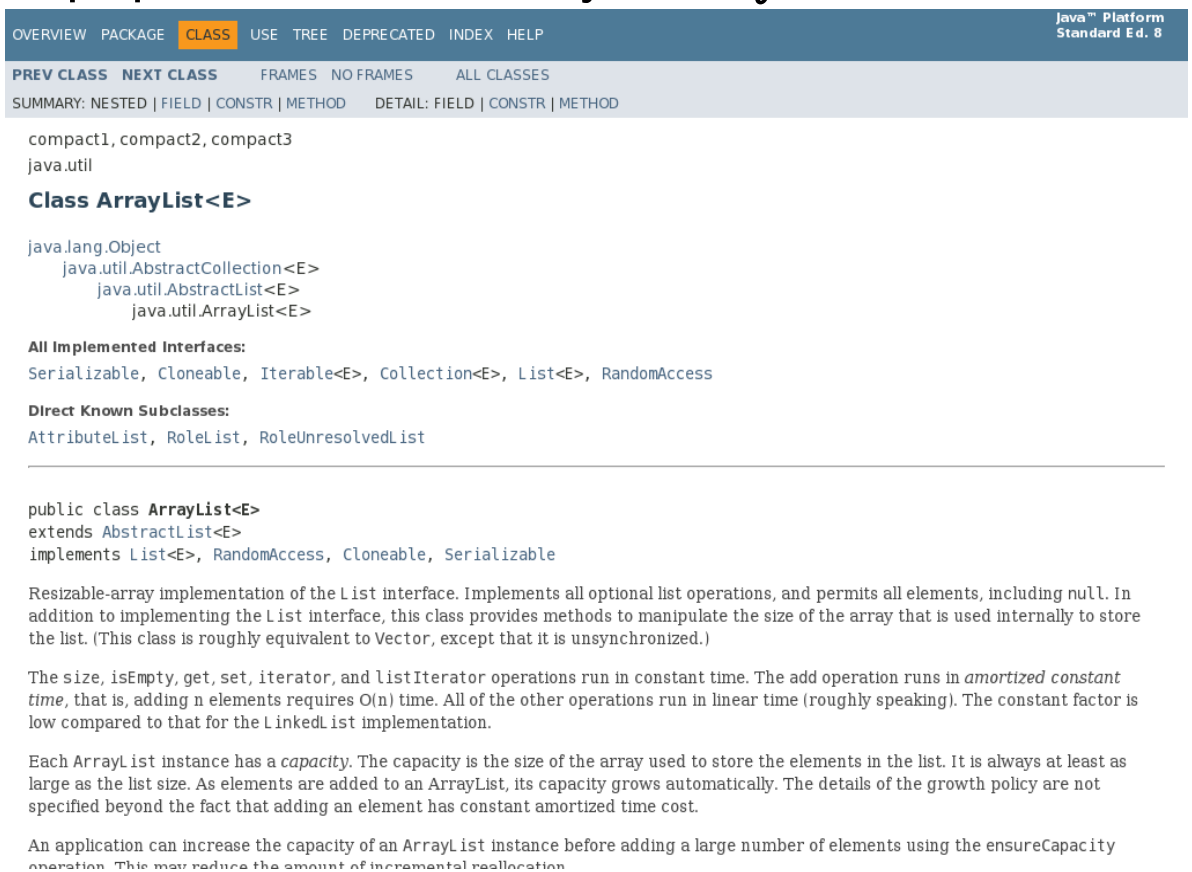
- Příklad výstupu:

```
java DemoSuper
This is B object
Constructor of class A
 Append in class B 'Text'
 Append in class A 'Text
```

lec08/DemoSuper.java

Hierarchie tříd – třída `ArrayList`

- V případě knihovní třídy `ArrayList`



OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP — Java™ Platform Standard Ed. 8

PREV CLASS NEXT CLASS FRAMES NO FRAMES ALL CLASSES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

compact1, compact2, compact3

java.util

Class ArrayList<E>

java.lang.Object
  java.util.AbstractCollection<E>
    java.util.AbstractList<E>
      java.util.ArrayList<E>

All Implemented Interfaces:

Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess

Direct Known Subclasses:

AttributeList, RoleList, RoleUnresolvedList

---

public class **ArrayList<E>**
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable

Resizable-array implementation of the List interface. Implements all optional list operations, and permits all elements, including null. In addition to implementing the List interface, this class provides methods to manipulate the size of the array that is used internally to store the list. (This class is roughly equivalent to Vector, except that it is unsynchronized.)

The size, isEmpty, get, set, iterator, and listIterator operations run in constant time. The add operation runs in *amortized constant time*, that is, adding n elements requires O(n) time. All of the other operations run in linear time (roughly speaking). The constant factor is low compared to that for the LinkedList implementation.

Each ArrayList instance has a *capacity*. The capacity is the size of the array used to store the elements in the list. It is always at least as large as the list size. As elements are added to an ArrayList, its capacity grows automatically. The details of the growth policy are not specified beyond the fact that adding an element has constant amortized time cost.

An application can increase the capacity of an ArrayList instance before adding a large number of elements using the ensureCapacity operation. This may reduce the amount of incremental reallocation

https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html

- Hierarchie nadtříd je větší.

V nadtřídách se objevuje také abstraktní třída.

Abstraktní třída

- Chceme-li předepsat, že odvozená třída musí implementovat metodu, specifikujeme tento požadavek klíčovým slovem **abstract**
- Abstraktní metody se mohou vyskytovat pouze v abstraktních třídách

  Jsou protikladem finálních metod, které nelze předefinovat.
- Abstraktní metody nemají implementaci

- Použití například pro vytvoření společného předka hierarchie tříd, které mají mít společné vlastnosti, případně doplněné o datové položky

Příklad abstraktní třídy

```
abstract class Geom {
    public abstract boolean intersect(Geom obj);
}

class Segment extends Geom {
    public boolean intersect(Geom obj) {
        ...
    }
}

class Circle extends Geom {
    public boolean intersect(Geom obj) {
        ...
    }
}
```

Rozhraní třídy – **interface**

- V případě potřeby „dědění" vlastností více předků lze využít rozhraní **interface**

  Řeší vícenásobnou dědičnost

- Rozhraní definuje množinu metod, které třída musí implementovat, pokud implementuje (**implemets**) dané rozhraní

  Garantuje, že daná metoda je implementována, neřeší však jak

- Rozhraní poskytuje specifický „pohled" na objekty dané třídy

  Můžeme přetypovat na objekt příslušného rozhraní

- Třída může implementovat více rozhraní

  Na rozdíl od dědění, u kterého může dědit pouze od jediného přímého předka

- Případnou „kolizi" shodných jmen metod více rozhraní řeší programátor

Příklad rozhraní

- Rozhraní definující schopnost objektu vykreslit se na plátno (canvas)

```
public interface Printable {
    public void printToCanvas(Canvas canvas);
}
```

- Geometrický objekt třídy `Segment` je odvozen od společné abstraktní třídy `Geom` a implementuje rozhraní `Printable`

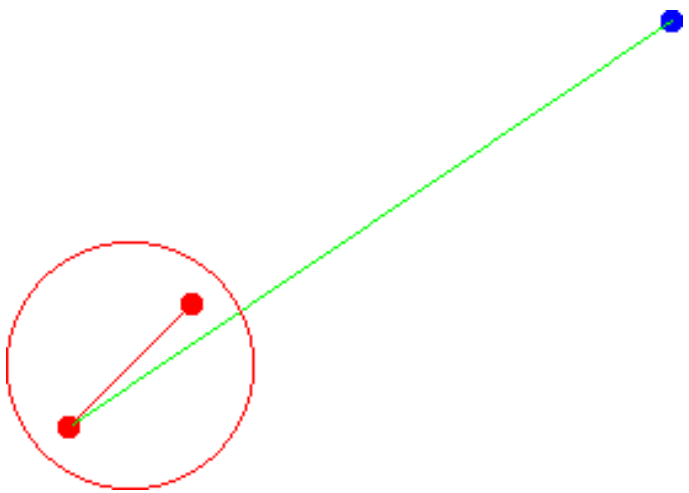
```
class Segment extends Geom implements Printable {
    @Override
    public boolean intersect(Geom obj) {
        ...
    }
    @Override
    public void printToCanvas(Canvas canvas) {
        ...
} }
```

Část II

Příklad geometrických objektů a jejich vizualizace

Zadání problému

- Cílem je vytvořit datovou reprezentaci základních geometrických objektů (bod, úsečka, kružnice, . . .) a relačních operací nad těmito objekty, např. zdali se některý objekt nachází uvnitř jiného objektu
- Dále chceme objekty umět vizualizovat v rastrovém obrázku
- Příklad: zobrazení všech objektů uvnitř kružnice červeně



Co je k dispozici

- Elementární rozhraní pro bod v rovině **Coords**
- Zobrazení:
  - Rozhraní plátna (**Canvas**) a základní implementace realizována polem polí (**ArrayBackedCanvas**)
  - Základní rasterizační funkce pro vykreslení úsečky a kružnice na mřížce (**GridCanvasUtil**)
  - Rozhraní pro zobrazitelné objekty **Printable**
  - Rozhraní pro uchování zobrazitelných objektů **ObjectHolder** a jeho základní implementace **ObjectHolderImpl**
- Elementární geometrické funkce pro testování, zdali jsou tři body kolineární a jestli bod leží na úsečce (**GeomUtil**)

- Implementace je součástí knihovny `lec08-simple_gui.jar`

Rozhraní Coords

- Bod v rovině je dán souřadnicemi x a y
- Rozhraní je dostatečné obecné, abychom jej mohli použít jak pro geometrický bod, tak pro pozici v mřížce obrázku
- Potřebujeme umět vytvořit bod a přečíst hodnoty x a y

```
public interface Coords {

   public int getX();
   public int getY();
   public Coords createCoords(int x, int y);

}
```

Pro jednoduchost uvažujeme pouze celá čísla

Rozhraní Canvas

- Plátno (canvas) má své rozměry, které potřebujeme znát

  Například abychom nevykreslovali mimo rozsah rastrového obrázku!

- Kreslení do rastrového obrázku nám postačí pouze změna barvy pixelu na příslušném políčku
- Pro barvu využijeme třídy **Color** z JDK

```
import java.awt.Color;

public interface Canvas {

    public int getWidth();
    public int getHeight();
    public void setColorAt(int x, int y, Color color);

}
```

Rozhraní Printable

- Od rozhraní **Printable** požadujeme pouze jedinou vlastnost a to umět se vykreslit na plátno (**Canvas**)

```
public interface Printable {

    public void printToCanvas(Canvas canvas);

}
```

- Způsob jakým se objekt vykreslí je závislý na konkrétním geometrickém objektu

Zde neřešíme a ani nemůžeme, protože nevíme jaké geometrické objekty budou definovány.

Rozhraní ObjectHolder

- Rozhraní **ObjectHolder** deklaruje metody pro přidání objektu a vykreslení všech uložených objektů

```
public interface ObjectHolder {

    public ObjectHolder add(Printable object);
    public void printToCanvas(Canvas canvas);

}
```

- Pořadí vykreslení v rozhraní neřešíme

Základní implementace ObjectHolderImpl

- Pro jednoduchou implementaci vystačíme s před alokovaným polem pro uložení zobrazitelných objektů

```
public class ObjectHolderImpl implements ObjectHolder {
   private final Printable[] objects;
   private int size; // the number of stored objects

   public ObjectHolderImpl(int max) {
      objects = new Printable[max];
      size = 0;
   }
   @Override
   public ObjectHolder add(Printable object) {
      if (object != null && size < objects.length) {
         objects[size++] = object;
      }
      return this;
   }
   @Override
   public void printToCanvas(Canvas canvas) {
      for (int i = 0; i < size; ++i) {
         objects[i].printToCanvas(canvas);
      }
   }
}
```

Knihovna rasterizačních funkcí GridCanvasUtil

- Využijeme rozhraní **Coords**
- Metody představují sadu utilit, proto volíme statické metody

```
public class GridCanvasUtil {
   private GridCanvasUtil() {} // library, no instance allowed

   /**
    * Bresenham's line algorithm to raster a straight-line segment
    into a grid
    * http://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm
    *
    * @param p0
    * @param p1
    * @return array of Coords representing rasterized segment from
    p0 to p1
    */
   public static Coords[] drawGridLine(Coords p0, Coords p1) {
      ...
   }

   public static Coords[] drawGridCircle(Coords ct, int radius) {
      ...
   }
}
```

Knihovna geometrických funkcí GeomUtil

- Podobně pro geometrické funkce v rovině

```
public class GeomUtil {
   private GeomUtil() {}
   /**
    * Compute the winding number
    *
    * @param a point forming a line
    * @param b point forming a line
    * @param c testing point for the wind number
    * @return 0 if c is on the line a-b, <0 if c is on the left of
    the line
    * a-b, >0 if c is on the right of the line a-b
    */
   public static int wind(Coords a, Coords b, Coords c) { ... }
   /**
    *
    * @param a
    * @param b
    * @param c
    * @return true if point c is in between points a and b
    */
   public static boolean inBetween(Coords a, Coords b, Coords c) {
     ... }
}
```

Jednoduchá realizace plátna – ArrayBackedCanvas

- Plátno je dvourozměrné pole barev

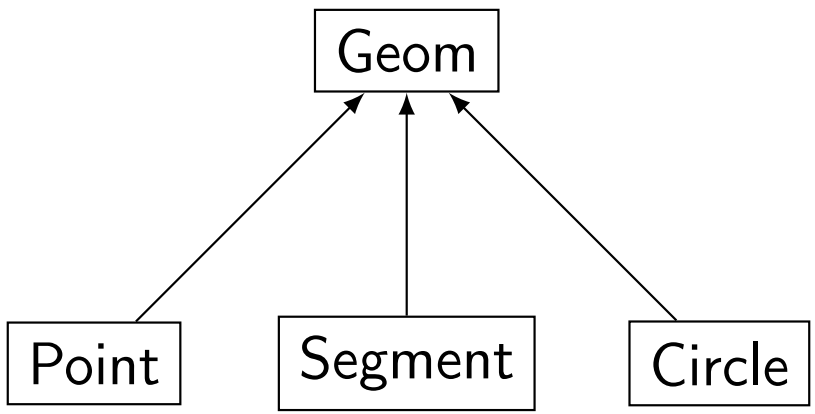
```
public class ArrayBackedCanvas implements Canvas {
    private final int width, height;
    private final Color[][] canvas;

    public ArrayBackedCanvas(int width, int height) {
        this.width = width;
        this.height = height;
        canvas = new Color[width][height];
        clearCanvas(Color.WHITE);
    }

    public void clearCanvas(Color color) { ... }

    @Override
    public void setColorAt(int x, int y, Color color) {
        canvas[x][y] = color;
    }

    public void writeToFile(String fileName) throws IOException {
       ...
    }
    private BufferedImage generateBufferedImage() { ... }
}
```

Návrh řešení

- Řešení založíme na rozhraní z balíku `lec08-simple_gui.jar` jehož implementace nám zajistí, že bude moci použít plátno pro zobrazení geometrických objektů
- Zároveň se pokusíme „oddělit" vizualizaci od vlastních geometrických operací, proto se nejdříve zaměříme na geometrické objekty
- Geometrické objekty však anotujeme barvou
  - Tedy, každý geometrický objekt má kromě svého popisu také barvu

- Postup návrhu
  1. Návrh hierarchie tříd geometrických objektů
  2. Návrh „testovací" funkce pro ověření funkčnosti
  3. Rozšíření návrhu o vizualizaci

Hierarchie tříd geometrických objektů



Abstraktní třída **Geom** 1/2

- Základní geometrický objekt představuje společnou abstraktní třídu
- Deklarujeme dvě základní geometrické operace **isEqual** a **isInside**

isEqual použijeme pro předefinování metody equals

```
public abstract class Geom {

   protected Color color;

   public Geom(Color color) {
      this.color = color;
   }

   public abstract boolean isEqual(Geom geom);
   public abstract boolean isInside(Geom geom);

   public Color getColor() {
      return color;
   }

   public void setColor(Color color) {
      this.color = color;
   }
}
```

Abstraktní třída **Geom** 2/2

- Dále předefinuje metody třídy `Object`

```
import java.util.Objects;

public abstract class Geom {
   @Override
   public String toString() {
      return "Geom{shape="+ getShapeName() + ",color=" + color + '}';
   }
   // @return string representation of the shape name
   public abstract String getShapeName();
   @Override
   public int hashCode() { ... }
   @Override
   public boolean equals(Object obj) {
      if (obj == null) { return false; }
      if (getClass() != obj.getClass()) { return false; }
      final Geom other = (Geom) obj;
      if (!Objects.equals(this.color, other.color)) {
         return false;
      }
      return isEquals(other); // isEquals is defined in derived class
   }
}
```

Třída **Point** 1/3

- Pro třídu **Point** připravíme několik konstruktorů
- **Point** také použijeme pro implementaci rozhraní **Coords** používané v metodách knihovny **GeomUtil**

```
public class Point extends Geom implements Coords {
    private final int x; // for simplicity we use int
    private final int y; // as coords in plane

    public Point(int x, int y) {
        this(x, y, Color.BLUE);
    }
    public Point(int x, int y, Color color) {
        this(x, y, color);
    }
    public Point(int x, int y, Color color) {
        super(color);
        this.x = x;
        this.y = y;
    }
    @Override
    public String getShapeName() {
        return "Point";
    }
```

Třída **Point** 2/3

- Rozhraní **Coords** předepisuje metody `getX()`, `getY()` a `createCoords()`

```
public class Point extends Geom implements Coords {
   ...
   @Override
   public int getX() {
      return x;
   }

   @Override
   public int getY() {
      return y;
   }

   @Override
   public Coords createCoords(int x, int y) {
      return new Point(x, y);
   }
   ...
}
```

Třída **Point** 3/3

- Implementace geometrických operací je omezena pouze na relace s jiným bodem

```
public class Point extends Geom implements Coords {

    @Override
    public boolean isEquals(Geom geom) {
        boolean ret = geom == this;
        if (!ret && geom instanceof Point) {
            Point pt = (Point) geom;
            ret = x == pt.x && y == pt.y;
        }
        return ret;
    }

    @Override
    public boolean isInside(Geom geom) {
        boolean ret = false; // A geom object cannot be inside a
      point
        return ret;
    }
}
```

Třída **Segment** 1/3

- Úsečka je definována dvěma body

```
public class Segment extends Geom {
   private final Point p0;
   private final Point p1;

   public Segment(Point pt1, Point pt2) {
      this(pt1, pt2, Color.GREEN);
   }

   public Segment(Point pt1, Point pt2, Color color) {
      super(color);
      if (pt1 == null || pt2 == null || pt1.equals(pt2)) {
         throw new IllegalArgumentException();
      }
      p0 = pt1;
      p1 = pt2;
   }
   @Override
   public String getShapeName() {
      return "Segment";
   }
}
```

Třída **Segment** 2/3

- Implementace geometrických operací je vztažena na **Point** i **Segment**

```
public class Segment extends Geom {
    ...
    @Override
    public boolean isInside(Geom geom) {
        if (geom == null) {
            return false;
        }
        boolean ret = this == geom;
        if (!ret && geom instanceof Point) {
            ret = isInside((Point) geom);
        } else if (!ret && geom instanceof Segment) {
            ret = isInside((Segment) geom);
        }
        return ret;
    }
    ...
}
```

Třída **Segment** 3/3

- Pro testování, zdali bod leží na úsečce, využijeme funkce z **GeomUtil**

```
public class Segment extends Geom {
   ...
   public boolean isInside(Point pt) {
      if (pt == null) {
         return false;
      }
      boolean collinear = GeomUtil.wind(p0, p1, pt) == 0;
      return collinear && GeomUtil.inBetween(p0, p1, pt);
   }
   public boolean isInside(Segment s) {
      if (s == null) {
         return false;
      }
      return isInside(s.p0) && isInside(s.p1);
   }
   ...
}
```

Třída **Circle** 1/3

- Pro kružnice volíme základní barvu červenou

```
public class Circle extends Geom {
   private final Point center;
   private final int radius;

   public Circle(Point center, int radius) {
      this(center, radius, Color.RED);
   }
   public Circle(Point center, int radius, Color color) {
      super(color);
      if (center == null || radius <= 0) {
         throw new IllegalArgumentException();
      }
      this.center = center;
      this.radius = radius;
   }
   @Override
   public String getShapeName() {
      return "Circle";
   }
}
```

Třída **Circle** 2/3

- Pro test **isInside** rozlišujeme už tři objekty

```
public class Circle extends Geom {
    ...
    @Override
    public boolean isInside(Geom geom) {
        if (geom == null) {
            return false;
        }
        boolean ret = this == geom;
        if (ret) {
            return ret;
        }
        if (geom instanceof Point) {
            ret = isInside((Point) geom);
        } else if (geom instanceof Segment) {
            ret = isInside((Segment) geom);
        } else if (geom instanceof Circle) {
            ret = isInside((Circle) geom);
        }
        return ret;
    }
    ...
}
```

Třída **Circle** 3/3

- Testujeme bod, úsečku a také jinou kružnici

```
public class Circle extends Geom {
   ...
   public boolean isInside(Point pt) {
      if (pt == null) { return false; }
      int dx = pt.getX() - center.getX();
      int dy = pt.getY() - center.getY();
      return ((dx * dx + dy * dy) <= radius * radius);
   }

   public boolean isInside(Segment sg) {
      if (sg == null) { return false; }
      return isInside(sg.getP0()) && isInside(sg.getP1());
   }

   public boolean isInside(Circle c) {
      if (c == null) { return false; }
      int rd = radius - c.radius;
      if (rd > 0) {
         return new Circle(center, rd).isInside(c.center);
      }
      return false;
   }
```

Příklad použití

- Zjištění, zdali testovací body leží uvnitř kružnice

```
Point pt1 = new Point(320, 240);
Circle c1 = new Circle(new Point(100, 100), 50);
Point pt2 = new Point(75, 75);
Point pt3 = new Point(125, 125);

Segment s1 = new Segment(pt1, pt2);
Segment s2 = new Segment(pt2, pt3);

System.out.println("pt1: " + pt1);

System.out.println("pt1 is inside circle: " + c1.isInside(pt1));
System.out.println("pt2 is inside circle: " + c1.isInside(pt2));

System.out.println("s1 is inside circle: " + c1.isInside(s1));
System.out.println("s2 is inside circle: " + c1.isInside(s2));
```

- Příklad výstupu

```
pt1: Geom{shape=Point,color=java.awt.Color[r=0,g=0,b=255]}
pt1 is inside circle: false
pt2 is inside circle: true
s1 is inside circle: false
s2 is inside circle: true
```

Ověření funkčnosti knihovny

- Pečlivě navrhneme konfigurace, pro které ověříme, že implementované řešení dává správný výsledek
- Hodnoty můžeme vypsat na standardní výstup nebo program „krokovat"
- Oba způsoby jsou sice funkční, ale přehlednější bude zobrazit výstup graficky

- Jednotlivým geometrickým objektům proto implementujeme rozhraní **Printable**, tj. rozšíříme je o metodu **printToCanvas**

Třída **Point** jako **Printable** 1/2

- Bod budeme vykreslovat nikoliv jako jeden pixel, ale jako malý disk o poloměru `radius`

Doplníme položku a rozšíříme konstruktor

```
public class Point extends Geom implements Coords,
    Printable {
  ...
  private int radius;

  public Point(int x, int y, Color color, int radius)
    {
      super(color);
      this.x = x;
      this.y = y;
      this.radius = radius;
  }
  ...
```

Třída **Point** jako **Printable** 2/2

- Implementujeme vykreslení disku o poloměru `radius`

```
public void printToCanvas(Canvas canvas) {
    if (canvas == null) { return; }
    final int w = canvas.getWidth();
    final int h = canvas.getHeight();
    final int r2 = radius * radius;
    for (int i = x - radius; i <= x + radius; ++i) {
        for (int j = y - radius; j <= y + radius; ++j) {
            if (i >= 0 && i < w && j >= 0 && j < h) {
                final int dx = (x - i);
                final int dy = (y - j);
                final int r = dx * dx + dy * dy;
                if (r < r2) {
                    canvas.setColorAt(i, j, color);
            } }
} } }
```

Třída **Segment** jako **Printable**

- Implementujeme vykreslení s využitím rasterizační funkce **drawGridLine** z `GridCanvasUtil`

```
public class Segment extends Geom implements Printable {
  @Override
  public void printToCanvas(Canvas canvas) {
    if (canvas == null) { return; }
    Coords[] line = GridCanvasUtil.drawGridLine(p0, p1);
    if (line == null) { return; }
    final int w = canvas.getWidth();
    final int h = canvas.getHeight();

    for (int i = 0; i < line.length; ++i) {
      Coords pt = line[i];
      if (
          pt.getX() >= 0 && pt.getX() < w &&
          pt.getY() >= 0 && pt.getY() < h
        ) {
        canvas.setColorAt(pt.getX(), pt.getY(), color);
      }
    }
} }
```

Třída **Circle** jako **Printable**

- Implementujeme vykreslení s využitím rasterizační funkce **drawGridLine** z `GridCanvasUtil`

```
public class Circle extends Geom implements Printable {
  @Override
  public void printToCanvas(Canvas canvas) {
    if (canvas == null) { return; }
    Coords[] pts =
       GridCanvasUtil.drawGridCircle(center, radius);
    if (pts == null) { return; }
    final int w = canvas.getWidth();
    final int h = canvas.getHeight();

    for (int i = 0; i < pts.length; ++i) {
      Coords pt = pts[i];
      if (
            pt.getX() >= 0 && pt.getX() < w &&
            pt.getY() >= 0 && pt.getY() < h
          ) {
          canvas.setColorAt(pt.getX(), pt.getY(), color);
    } }
} }
```

Příklad vykreslení objektů

- Geometrické objekty se už umí vykreslit na plátno (canvas)
- Vytvoříme instanci **ArrayBackedCanvas**
- „Zašleme" zprávu příslušnému objektu, aby se vykreslil
- Obsah plátna následně uložíme do souboru

```
Circle c1 = new Circle(new Point(100, 100), 50);
ArrayBackedCanvas canvas =
    new ArrayBackedCanvas(640, 480);

c1.printToCanvas(canvas);
canvas.writeToFile("circle.png");
```

Vykreslená kružnice v souboru `circle.png`

Další úkoly

- Máme implementovány základní funkčnosti pro zobrazení
- Vykreslení objektů však není příliš pohodlné
- Vytvoříme proto „kontejner" pro reprezentaci scény a hromadnější dotazy, zdali jsou objekty scény uvnitř zvoleného geometrického objektu

- Scénu realizujeme jako třídu **GeomObjectArray**, která bude poskytovat pole aktuálních objektů

```
public class GeomObjectArray {
    ...
    public Geom[] getArray() { ... }
    ...
}
```

Dokončení příště

Shrnutí přednášky

Diskutovaná témata

- Třídy a objekty
- Metoda `main`
- Objekty základních typů
- Immutable objekty
- Dědičnost a hierarchie tříd
- Příklad geometrických objektů, jednoduchých operací s nimi a jejich vykreslení
- Příště: Dokončení příkladu, kompozice objektů, balíčky, kompilace a spouštění Java programu