

# Objektově orientované programování

Jan Faigl

Katedra počítačů  
Fakulta elektrotechnická  
České vysoké učení technické v Praze

Přednáška 9

**A0B36PR1 – Programování 1**

## Část 2 – Příklad tříd geometrických objektů a jejich vizualizace

Přehled úlohy

Řešení z 8. přednášky

Příklad použití

## Část 1 – Objektově orientované programování - Dědičnost, kompozice a balíky

OOP

Dědičnost - návrh

Kompozice

Objektově orientovaný návrh programu

Organizace tříd

## Část I

### Objektově orientované programování - Dědičnost, kompozice a balíky

## Objektově orientované programování (OOP)

- **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 zasláním 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.

### Příklad – Kvádr je rozšířený **obdélník**?

```
class Rectangle {
    protected double width;
    protected double height;
    Rectangle(int width, int height) {
        this.width = width;
        this.height = height;
    }
    public double getWidth() {
        return width;
    }
    public double getHeight() {
        return height;
    }
    public double getDiagonal() {
        return Math.sqrt(width*width + height*height);
    }
}
```

## 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 k znovupoužitelnosti programového kódu

*Spolu s principem zapouzdření.*
- Dědičnost je základem polymorfismu

### Příklad – **Kvádr** je rozšířený obdélník?

```
class Cuboid extends Rectangle {
    protected double depth;

    Cuboid(int width, int height, int depth) {
        super(width, height); //konstruktor predka
        this.depth = depth;
    }
    public double getDepth() {
        return depth;
    }
    @Override
    public double getDiagonal() {
        double tmp = super.getDiagonal(); //volani predka
        return Math.sqrt(tmp*tmp + depth*depth);
    }
}
```

## Příklad dědičnosti – 1/2

- Třída **Cuboid** je rozšířením třídy **Rectangle** o hloubku (**depth**)
- Potomka deklarujeme rozšířením **extends**
  - Cuboid přebírá datové položky **width** a **height**
  - Cuboid také přebírá „gettery“ **getWidth** a **getHeight**
  - Konstruktor se nedědí, lze ale volat v podtřídě operátorem **super**
  - Není-li konstruktor deklarován, je volán konstruktor bez parametrů
    - Konstruktor existuje vždy, buď implicitní nebo uživatelský
- Potomek doplňuje datové položky o **depth** a mění metodu **getDiagonal**

## Dědičnost – Kvádr je rozšířený obdélník

- V příkladu jsme rozšiřovali obdélník a vytvořili „specializaci“ kvádr

**Je to skutečně vhodné rozšíření?**

Jaká je plocha kvádrů? Jaký je obvod kvádrů?

## Příklad dědičnosti – 2/2

- Objekty třídy **Cuboid** mohou využívat proměnné **width**, **height** a **depth**
- Metoda **getDiagonal** překrývá původní metodu definovanou v nadřazené třídě **Rectangle**

*zastínění – overriding*
- Přístup k původní metodě předka je možný přes operátor **super**
- Má-li metoda stejného jména jiné parametry (počet/typ) jedná se o **přetížení – overloading**

*Jedná se o jinou (novou) metodu!*

Dědičnost – Obdélník je speciální **kvádr**?

- Obdélník je kvádr s nulovou hloubkou
- ```

class Cuboid {
    protected double width;
    protected double height;
    protected double depth;
    Cuboid(int w, int h, int d) {
        this.width = w; this.height = h; this.depth = d;
    }
    public double getWidth() { return width; }
    public double getHeight() { return height; }
    public double getDepth() { return depth; }
    public double getDiagonal() {
        double tmp =
            width*width + height*height + depth*depth;
        return Math.sqrt(tmp);
    }
}

```

## Dědičnost – Obdélník je speciální kvádr?

```
class Rectangle extends Cuboid {
    Rectangle(int width, int height) {
        super(width, height, 0);
    }
}
```

- Obdélník je „kvádrem“ s nulovou hloubkou
- Potomek se deklaruje klíčovým slovem **extends**
  - **Rectangle** přebírá všechny datové položky **width**, **height** a **depth**
  - a také přebírá všechny metody předka (přístupné mohou být, ale pouze některé)
  - Konstruktor je přístupný přes volání **super** a hodnota proměnné **depth** se nastavujeme na nulu
- Objekty třídy **Rectangle** mohou využívat všech proměnných a metod

## Vztah předka a potomka je typu „is-a“

- Je úsečka potomkem bodu?
  - Úsečka nevyužije ani jednu metodu bodu
  - **is-a?:** úsečka je bod? → **NE** → úsečka není potomkem bodu
- Je obdélník potomkem úsečky?
  - **is-a?:** **NE**
- Je obdélník potomkem čtverce nebo naopak?
  - Obdélník rozšiřuje čtverec o další rozměr, ale není čtvercem
  - Čtverec je obdélník, který má šířku a výšku stejnou

*Nastavení délek stran v konstruktoru!*

## Je obdélník potomek kvádru nebo kvádr potomek obdélníka?

1. Kvádr je potomek obdélníka
  - Logické přidání rozměru, ale metody platné pro obdélník nefungují pro kvádr

*obsah obdélníka*
2. Obdélník je potomek kvádru
  - Logicky správná úvaha o specializaci: „vše co funguje pro kvádr funguje i pro kvádr s nulovou hloubkou“
  - Neefektivní implementace – každý obdélník je reprezentován 3 rozměry

### Specializace je správná

*Vše co platí pro předka, musí platit pro potomka*

*V tomto konkrétním případě je však použití dědičnosti diskutabilní.*

## Substituční princip

- Vzájemný vztah mezi dvěma odvozenými třídami
- Zásady:
  - Odvozená třída je specializací nadřazené třídy

*Existuje vztah is-a*

- Všude, kde lze použít třídu musí být použitelný i její potomek a to tak, aby uživatel nepoznal rozdíl

*Polymorfismus*

- Vztah **is-a** musí být trvalý

## Kompozice objektů

- Obsahuje-li deklarace třídy členské proměnné objektového typu, pak se jedná o **kompozici objektů**
- Kompozice vytváří hierarchii objektů – nikoliv však dědičnost  
*Dědičnost vytváří také hierarchii vztahů, ale ve smyslu potomek/předek.*
- Kompozice je vztah objektů **agregace – je tvořeno / je součástí**
- Jedná se o strukturu typu „has“

## Příklad kompozice 2/3

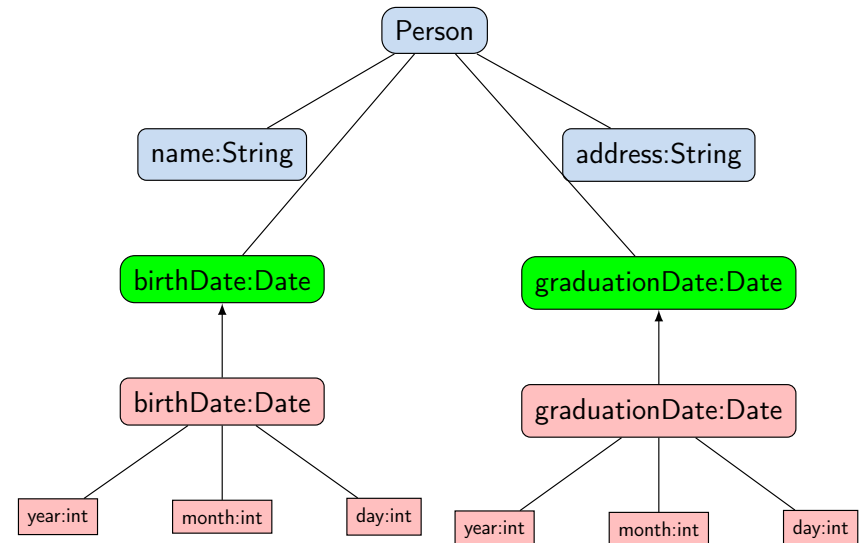
```
class Person {
    String name;
    String address;
    Date birthDate;
    Date graduationDate;
}
```

```
class Date {
    int day;
    int month;
    int year;
}
```

## Příklad kompozice 1/3

- Každá osoba je charakterizována atributy třídy Person
  - Jméno – name
  - Adresa – address
  - Datum narození – birthDate
  - Datum úspěšného ukončení studia – graduationDate
- Datum je charakterizováno třemi atributy (třída Date)
  - Den – day (int)
  - Měsíc – month (int)
  - Rok – year (int)

## Příklad kompozice 3/3



## Dědičnost vs kompozice

- Vlastnosti dědění objektů:
  - Vytváření odvozené třídy (potomek, podtřída)
  - Podtřída se vytváří **extends**
  - Odvozená třída je specializací nadřazené třídy
    - Přidává proměnné *Nebo také překrývá proměnné*
    - Přidává nebo modifikuje metody
  - Na rozdíl od kompozice mění vlastnosti objektů
    - Nové nebo modifikované metody
    - Přístup k proměnným a metodám předka (bázové třídě, supertřídě)
 

*Pokud je přístup povolen (public/protected/„package“)*
- Kompozice objektů je tvořena atributy objektového typu
 

*Pouze skládá objekty*
- Rozlišení mezi kompozicí nebo děděním (pomůcka)
  - „Je“ test – příznak dědění (is-a)
  - „Má“ test – příznak kompozice (has)

## Objektově orientovaný návrh programu

- Vychází z objektového přístupu k řešení problému
- Vytváří objektový model problému
- Můžeme rozdělit na následující kroky
  1. Formulace problému
  2. Návrh výpočetního řešení (algoritmu)
  3. Předběžný návrh architektury
  4. Podrobný návrh architektury

## Dědičnost a kompozice – úskalí

- Přílišné používání kompozice i dědičnosti v případech, kdy to není potřeba vede na příliš složitý návrh
- Pozor na doslovné výklady vztahu **is-a** a **has**, někdy se nejedná ani o dědičnost, ani kompozici
 

*Např. Point2D a Point3D nebo Circle a Ellipse*
- Dáváme přednost kompozici před dědičností
 

*Jedna z výhod dědičnosti je **polymorfismus***
- Při používání dědičnosti dochází k porušení zapouzdření
 

*Zejména s nastavením přístupových práv **protected***

## Formulace problému

- Definujeme:
  - Požadavky nutné pro řešení problému
  - Formu výsledku
- Dále volíme další postup návrhu algoritmu a jeho implementaci
 

*Nevhodnou formulací můžeme výrazně ztížit proces nalezení vhodného řešení. V extrémním případě takové řešení ani nemusí existovat.*

## Návrh algoritmu

Volíme na základě dvou faktorů

### 1. Typ řešeného problému

- Existence exaktního řešení
- Požadavek přesného nebo přibližného řešení
- Možnost dekompozice problému
- Velikost množiny vstupních dat
- Přesnost vstupních dat

### 2. Požadavku na výsledky

- Přesnost řešení (chyba modelu, chyba výstupních dat)
- Časová složitost
- Paměťová složitost

- Postup návrhu nazýváme **strategií implementace**

## Podrobný návrh architektury

- Uplatňujeme strategii zjemňování
- Návrh opakovaně upravujeme a zpřesňujeme
- Postup představuje aplikaci přístupu „shora dolů”

*Při návrhu zacházíme do čím dál hlubších podrobností.*

- Případné nedostatky nebo neefektivní konstrukce nahrazujeme vhodnějšími postupy

*Opakované návrhy nám pomáhají lépe pochopit problém a navrhnout vhodné řešení. I kompletní náhrada již „hotového” řešení může být někdy vhodná. Nebojte se „zahodit” nějaký návrh ve prospěch nového a lepšího řešení. To je nedílnou součástí procesu řešení problému.*

- Návrh má 3 základní fáze:

1. Návrh tříd
2. Návrh datových položek
3. Návrh metod

## Předběžný návrh architektury

- Úvaha o rozčlenění problému na třídy
- Operace, které mohou být s instancemi tříd prováděny
- Návrh rozhraní pro komunikaci mezi objekty (instancemi třídy)

*V této fázi návrhu může být vhodné využít grafický návrh pro názornou představu závislostí mezi komponentami – diagramy, tabulky.*

## Postup návrhu tříd

1. Stanovíme jednotlivé třídy, které budou sloužit k řešení dílčích podproblémů
2. Míra konkrétnosti podproblému
  - Třídy pro práci s daty (ukládání / načítání)
  - Třídy realizující výpočty
  - Třídy zajišťující interakci s uživatelem
  - Třída poskytující grafické rozhraní

*Oddělením zajistíme čitelnost a udržitelnost programu.*

- Komplexní návrh tříd představuje složitý problém
- Během procesu návrhu můžeme identifikovat:
  - Některé třídy mohou vykonávat duplicitní činnost
  - Činnosti, které nejsou vykonávány žádnou třídou
- **Postup návrhu tak několikrát opakujeme dokud nevznikne použitelný návrh**

## Návrh datových položek

- Jedná se o návrh jak budou uložena data
- Způsob uložení dat ovlivňuje efektivitu prováděných operací nad těmito daty
- Při návrhu zohledňujeme:
  - Volbu základních nebo uživatelských typů
  - Volbu dynamických datových struktur s ohledem na způsob řešení problému (přímý nebo sekvenční přístup)
  - Volbu vhodných datových typů vzhledem k požadované přesnosti vstupních a výstupních dat

## Vlastní návrh metod

1. Volba formálních parametrů metod
  2. Volba datových typů formálních parametrů metod
  3. Volba návratových typů metod
- Při řešení výpočetního problému zpravidla při návrhu zohledňujeme požadovanou přesnost dat
    - Přesnost operací prováděných s daty uvnitř jednoduchých metod musí být stejná nebo vyšší než přesnost, s jakou jsou data uložena.

## Návrh metod

- Metody popisují činnosti, které mohou být s daty prováděny
- Navrhujeme rozhraní objektů tříd a stanovujeme, které metody budou veřejné, a které soukromé
- Výsledkem návrhu je tabulka objektů a jejich metod znázorňující operace prováděné s jednotlivými objekty
- Zásady:
  - S každým objektem by měla být prováděna aspoň jedna operace
  - Každé operaci by měl odpovídat nějaký objekt, se kterým bude operace prováděna

*Pokud identifikujeme v návrhu metody a objekty, které tyto zásady nesplňují, je nutné provést korekci návrhu.*

## Balíky tříd

- V objektově orientovaném přístupu vzniká množství tříd (a rozhraní) a při použití několika knihoven se jen zřídka lze vyhnout kolizi unikátního jména třídy
- Proto organizujeme třídy do balíku tříd **package**, který reprezentuje ucelený soubor tříd, pro řešení nějakého problému
- Jméno balíku zapisujeme malými písmeny a ve složených názvech oddělujeme slova tečkou, např.

**cz.cvut.fel.pr1**

*Zvolit unikátní jméno balíku je zpravidla jednodušší než jméno třídy. Můžeme například odvodit jméno balíku od doménového jména instituce, které je samo o sobě unikátní.*

- **Plné jméno třídy** je tvořeno jménem balíku a jménem třídy, např.

**cz.cvut.fel.pr1.Lab07**



## Organizace tříd do balíků

- Třídy a rozhraní organizujeme do balíku
- V balíku jsou vždy umístěny související třídy
- Jednoznačná identifikace třídy je tvořena plně kvalifikovaným jménem třídy, které je složeno ze jména balíku a jména třídy
- Třídy ze stejného balíku jsou navzájem viditelné
- Při použití třídy z jiného balíku je třeba:
  - Použít plně kvalifikované jméno třídy
  - Importovat třídy klíčovým slovem **import** s uvedením plně kvalifikovaného jména třídy (případně balíku s hvězdičkovou notací).

*Jeden ze zásadních podpůrných nástrojů pro udržitelost velmi rozsáhlých (enterprise) programů.*

## Import třídy

- V Javě jsou automaticky přístupné třídy z balíku [java.lang](#)
- Ostatní třídy je nutné adresovat plně kvalifikovaným jménem nebo importovat
- Import provede zpřístupnění jmenného prostoru třídy

```
// zavedeni tridy TextIO
import cz.cvut.fel.pr1.TextIO;
```

```
// zavedeni celeho jmenneho prostoru
// cz.cvut.fel.pr1
import cz.cvut.fel.pr1.*;
```

## Organizace tříd do balíků

- Jméno balíku je složeno z dílčích jmen oddělených tečkami
- Fyzická reprezentace balíku je odpovídající adresářová struktura
- Na cvičení používáte balík [cz.cvut.fel.pr1](#)

## Příklad použití plně kvalifikovaného jména třídy

- Použití třídy **Scanner**

```
public class DemoPackage {
    public void start() {
        java.util.Scanner scan =
            new java.util.Scanner(System.in);
        System.out.print("Enter a line: ");
        String line = scan.nextLine();
        System.out.println("The length of the entered
        lines is: " + line.length());
    }
    public static void main(String args[]) {
        DemoPackage demo = new DemoPackage();
        demo.start();
    }
}
```

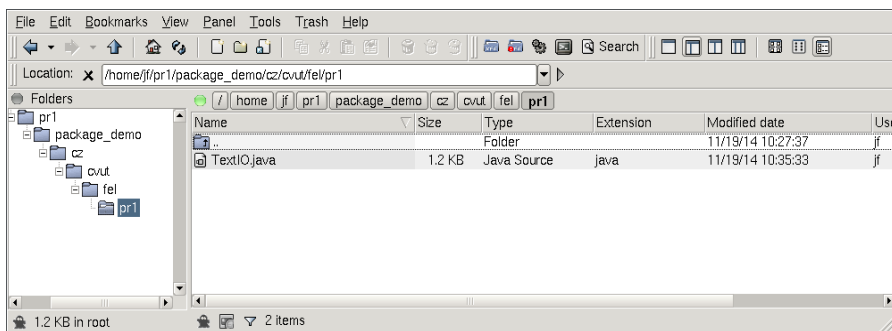
## Příklad importu třídy

```
import java.util.Scanner;

public class DemoPackage {
    public void start() {
        Scanner scan = new Scanner(System.in);
        System.out.print("Enter a line: ");
        String line = scan.nextLine();
        System.out.println("The length of the entered
lines is: " + line.length());
    }
    public static void main(String args[]) {
        DemoPackage demo = new DemoPackage();
        demo.start();
    }
}
```

## Kompilace a spuštění třídy z balíku 2/5

- Zdrojový soubor TextIO.java je v adresářové struktuře odpovídající jménu balíku



lec09/package\_demo

## Kompilace a spuštění třídy z balíku 1/5

- Napište jednoduchý kód demonstrující použití třídy TextIO
- Třída se nachází v balíku cz.cvut.fel.pr1
- Ve třídě implementujeme metodu **public static void main(String[] args)**
- Dále pak metodu **demo**, která načte řádek ze standardního vstupu a vypíše, zda-li řádek reprezentuje celé číslo nebo reálné číslo typu double

lec09/package\_demo

## Kompilace a spuštění třídy z balíku 3/5

```
package cz.cvut.fel.pr1;
...
public class TextIO {
    ...
    public static boolean isInteger(String s) { ... }
    public static boolean isDouble(String s) { ... }
    public void demo() {
        System.out.println("Demo of the TextIO class");
        System.out.print("Enter a line(number or a word): ");
        String line = getLine();
        if (isInteger(line)) {
            System.out.println("The line represents integer");
        } else if (isDouble(line)) {
            System.out.println("The line represents double");
        } else {
            System.out.println("The line is a general string");
        }
    }
    public static void main(String[] args) {
        TextIO txtIO = new TextIO();
        txtIO.demo();
    }
}
```

## Kompilace a spuštění třídy z balíku 4/5

- Při kompilaci specifikujeme jméno souboru pro překlad  
**javac cz/cvut/fel/pr1/TextIO.java**
  - Zkompilovaný soubor se nachází ve stejném adresáři  
**cz/cvut/fel/pr1/TextIO.class**  
*Jiný adresář lze předepsat přepínačem -d, viz javac -help.*
  - Při spuštění specifikujeme plné jméno třídy včetně jména balíku  
**java cz/cvut/fel/pr1/TextIO**  
*Standardně je prohledáván aktuální pracovní adresář.*
- lec09/package\_demo

## Část II

### Příklad geometrických objektů a jejich vizualizace

## Kompilace a spuštění třídy z balíku 5/5

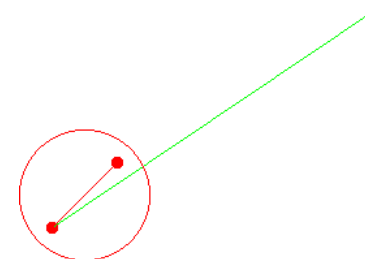
- Při jiném umístění třídy (zkompilovaného .class souboru) se nám nepodaří třídu spustit

```
% rm cz/cvut/fel/pr1/TextIO.class
% mkdir classes
% javac -d classes cz/cvut/fel/pr1/TextIO.java
% java cz.cvut.fel.pr1.TextIO
Error: Could not find or load main class cz.cvut
.fel.pr1.TextIO
% java -classpath classes cz.cvut.fel.pr1.TextIO
Demo of the TextIO class
Enter a line(number or a word):
```

lec09/package\_demo

## Zadání problému

- Popis scény geometrických objektů a určení zda-li objekty leží uvnitř jiného objektu
- Popis vektorových geometrických objektů
- Vizualizace geometrických objektů v rastrovém obrázku



## Výchozí knihovny a rozhraní

- 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**
- Knihovny funkci **GeomUtil** a **GeomCanvasUtil**

lec08-simple\_gui.jar

## Rozhraní Canvas

```
import java.awt.Color;

public interface Canvas {

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

}
```

## Rozhraní Coords

```
public interface Coords {

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

}
```

## Rozhraní Printable

```
public interface Printable {

    public void printToCanvas(Canvas canvas);

}
```

## 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);

}
```

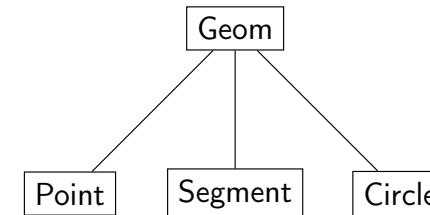
## Abstraktní třída Geom

```
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 abstract String getShapeName();

    public Color getColor() { ... }
    public void setColor(Color color) { ... }
    @Override
    public String toString() { ... }
    @Override
    public int hashCode() { ... }
    @Override
    public boolean equals(Object obj) { ... }
}
```

## Hierarchie tříd geometrických objektů



## Třída Point 1/2

```
public class Point extends Geom implements Coords,
    Printable {
    private final int x;
    private final int y;
    private int radius;

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

Třída `Point` 2/2

```
public class Point extends Geom implements Coords,
    Printable {
    ...
    @Override
    public String getShapeName() {
        return "Point";
    }

    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` 2/2

```
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 `Segment` 1/2

```
public class Segment extends Geom implements Printable {
    private final Point p0;
    private final Point p1;
    ...
    public Segment(Point pt1, Point pt2, Color color) {
        super(color);
        ...
    }
    ...
    @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 `Circle` 1/2

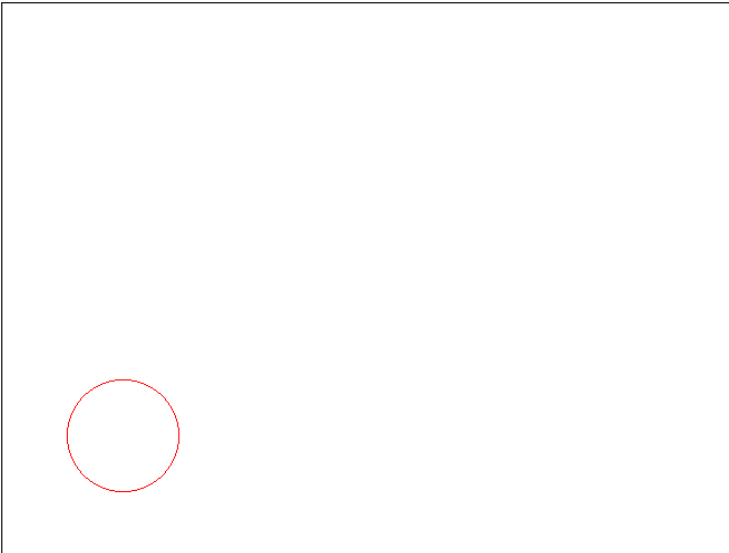
```
public class Circle extends Geom implements Printable {
    ...
    @Override
    public boolean isInside(Geom geom) {
        if (geom == null) {
            return false;
        }
        boolean ret = this == geom;
        if (ret) {
            return ret;
        }
        if (geom instanceof 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` 2/2

```
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);
            }
        }
    }
}
```

## Vykreslená kružnice v souboru `circle.png`



## 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");
```

## Další úkoly

- Máme implementovány základní funkčnosti pro zobrazení
- Vykreslení objektů však není příliš pohodlné
- Vytvoříme proto „kontejner” pro reprezentaci scény a hromadnější dotazy, zda-li jsou objekty scény uvnitř zvoleného geometrického objektu
- Scénu realizujeme jako třídu `GeomObjectArray`, která bude poskytovat pole aktuálních objektů

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

## GeomObjectArray

- „Kontejner“ realizujeme jako před-alokované pole „dostatečné velikost“
- Implementujeme metodu `add`, která přidá objekt do pole
- V případě naplnění kapacity alokujeme pole větší, kterým nahradíme pole původní

*Přístup k poli zapouzdříme metodami `add` a `getArray`*

```
public class GeomObjectArray {
    private Geom[] objects;
    private int size;
    private final int DEFAULT_SIZE_RESERVE = 100;
    public GeomObjectArray() {
        objects = new Geom[DEFAULT_SIZE_RESERVE];
    }
    public GeomObjectArray add(Geom obj) { ... }
    public Geom[] getArray() { ... }
}
```

## GeomObjectArray – getArray

```
public class GeomObjectArray {
    ...
    public Geom[] getArray() {
        Geom[] ret = new Geom[size];
        for (int i = 0; i < size; ++i) {
            ret[i] = objects[i];
        }
        return ret;
    }
}
```

*Na první pohled možná neefektivní, ale metoda nám zajišťuje, že je „scéna“ reprezentována polem o velikost odpovídající objektům ve scéně.*

## GeomObjectArray – add

```
public class GeomObjectArray {
    ...
    public GeomObjectArray add(Geom obj) {
        if (size == objects.length) {
            Geom[] objectsNew =
                new Geom[objects.length + DEFAULT_SIZE_RESERVE];
            for (int i = 0; i < objects.length; ++i) {
                objectsNew[i] = objects[i];
            }
            objects = objectsNew; //replace the array
        }
        objects[size++] = obj;
        return this; //return this to string add call
    }
    ...
}
```

## Obarvení objektů uvnitř jiného objektu

- Vytvoříme metodu, která nastaví barvu všech objektů uvnitř jiného objektu

```
public void markColorInside(Geom[] objects, Geom
    largeObject, Color color) {
    if (objects == null || largeObject == null || color
        == null) {
        return;
    }
    for (int i = 0; i < objects.length; ++i) {
        Geom obj = objects[i];
        if (obj != null && largeObject.isInside(obj)) {
            obj.setColor(color);
        }
    }
}
```



## Příklad použití

```
public void start(String[] args) {
    Point pt1 = new Point(320, 240);
    Point pt2 = new Point(75, 75);
    Point pt3 = new Point(125, 125);

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

    Circle c1 = new Circle(new Point(100, 100), 50);
    Circle c2 = new Circle(new Point(400, 400), 400);

    c2.setColor(new Color(255, 130, 71)); //RoyalBlue

    GeomObjectArray geomObjects = new GeomObjectArray();

    geomObjects.add(pt1).add(pt2).add(pt3).add(s1).add(s2);
    geomObjects.add(c1).add(c2);

    markColorInside(geomObjects.getArray(), c1, Color.RED);
}

pr1-lec09-demo_gui
```

## Uložení obrázku

- Uložení plátna můžeme spojit s překreslením aktuální scény v metodě `saveCanvas`

```
public void saveCanvas(GeomObjectArray geomObjects,
    ArrayBackedCanvas canvas, String out) {
    try {
        canvas.clearCanvas(Color.WHITE);
        printToCanvas(geomObjects.getArray(), canvas);
        canvas.writeToFile(out);
    } catch (IOException e) {
        System.err.println("Error: writing canvas to the
            file '" + out + "'");
    }
}
```

## Příklad použití – vykreslení

- Pro vykreslení můžeme použít objekt, který má rozhraní `ObjectHolder`
- Buď můžeme použít implementaci `ObjectHolderImpl` *Např. kompozicí*
- Nebo využijeme znalosti, že některé podtřídy `Geom` implementují rozhraní `Printable` a implementujeme metodu `printToCanvas`

```
public void printToCanvas(Geom[] objects, Canvas
    canvas) {
    for (int i = 0; i < objects.length; ++i) {
        Geom obj = objects[i];
        if (obj instanceof Printable) {
            ((Printable) obj).printToCanvas(canvas);
        } } }
```

Zkuste rozšířit třídu `GeomObjectArray` o rozhraní `Printable` a vysvětlete úskalí implementace dvou metod `add`.

## Použití metody `markColorInside`

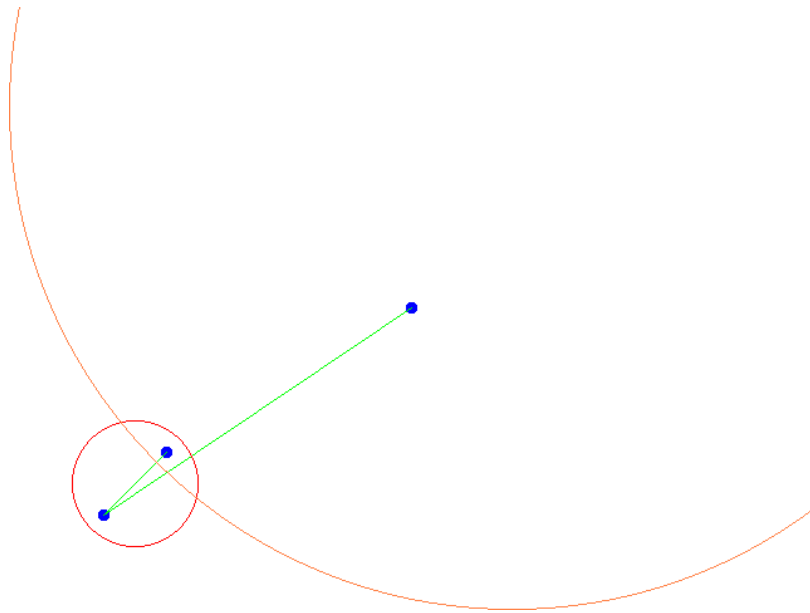
```
...
GeomObjectArray geomObjects = new GeomObjectArray();
geomObjects.add(pt1).add(pt2).add(pt3).add(s1).add(s2);
geomObjects.add(c1).add(c2);

ArrayBackedCanvas canvas = new ArrayBackedCanvas(640, 480);
saveCanvas(geomObjects, canvas, "objects.png");

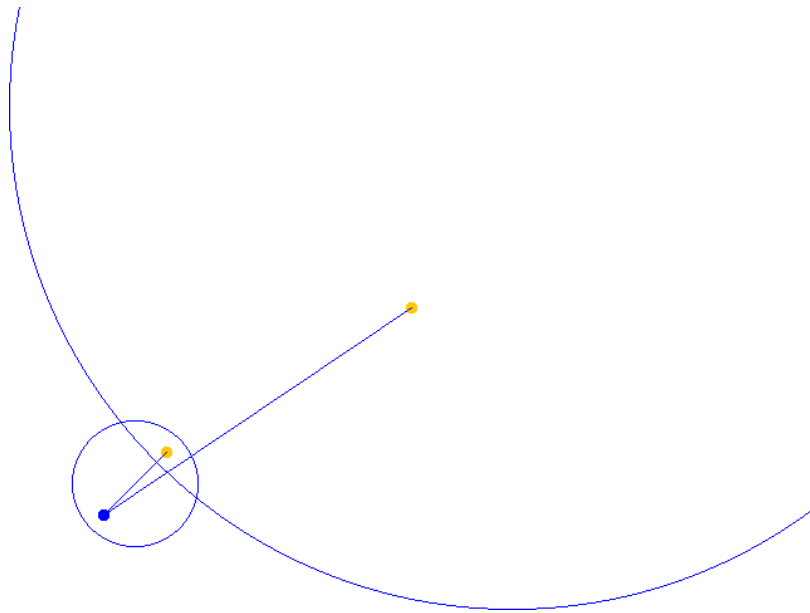
markColorInside(geomObjects.getArray(), c1, Color.RED);
saveCanvas(geomObjects, canvas, "objects-inside_circle-c1.png");

setColor(geomObjects.getArray(), Color.BLUE);
markColorInside(geomObjects.getArray(), c2, Color.ORANGE);
saveCanvas(geomObjects, canvas, "objects-inside_circle-c2.png");
```

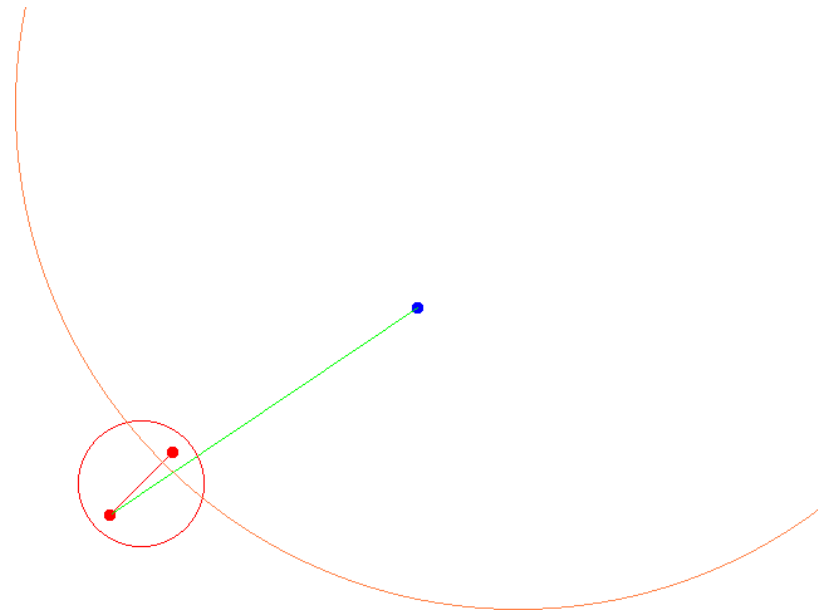
## Příklad výstupu – objects.png



## Příklad výstupu – objects-inside\_circle-c2.png



## Příklad výstupu – objects-inside\_circle-c1.png



## Příklad geometrický objektů

- Navrhli jsme hierarchii geometrický objektů
- Implementovali jsme operaci testování, zda-li objekt leží uvnitř jiného objektu `isInside`
- Vizualizaci jsem realizovali s využitím rozhraní a implementací z balíku třídy `lec08-simple_gui.jar`
- Použití jednotlivých objektů je možné, ale reprezentace scény je není pohodlná
- Pro práci s množinou objektů potřebujeme vhodný „kontejner“
- Zatím známe pouze datový typ **pole statické délky**
- Pro efektivní práci s více proměnnými potřebujeme složitější **datové struktury**

*Spojové seznamy, fronty, zásobníky, pole dynamické délky, atd.*

## Shrnutí přednášky

## Diskutovaná témata

- Objektivě orientované programování
- Dědičnost
- Kompozice
- Balíky – organizace tříd
- Kompilace a spuštění třídy v balíku
- Příklad geometrických objektů a jejich vykreslení (jednoduchý kontejner)
- **Příště: Spojový seznam a abstraktní datové typ**