Příklad tříd geometrických objektů a jejich vizualizace

Jiří Vokřínek

Katedra počítačů

Fakulta elektrotechnická

České vysoké učení technické v Praze

Přednáška 2

B6B36PJV – Programování v JAVA

Příklad tříd geometrických objektů a jejich vizualizace

Zadání

Popis výchozích rozhraní a tříd

Návrh řešení

Implementace

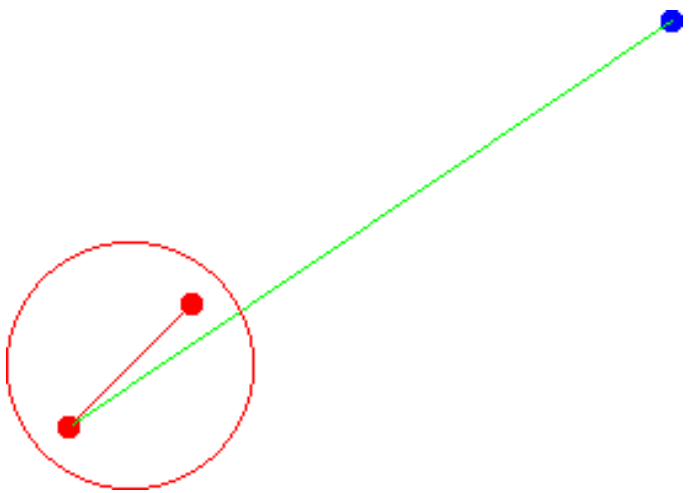
Příklad použití

Část I

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 `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 `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

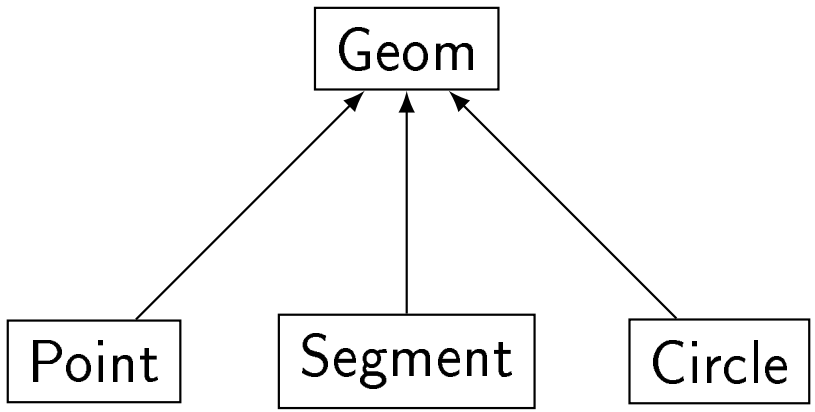
- Postup návrhu
  1. Návrh hierarchie tříd geometrických objektů
  2. Návrh „testovací" funkce pro ověření funkčnosti
  3. Rozšíření návrhu o vizualizaci

Návrh řešení

- Řešení založíme na rozhraní z balíku `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**

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() { ... }
    ...
}
```

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() { ... }
    ...
}
```

GeomObjectArray

- „Kontejner" realizujeme jako před-alokované pole „dostatečné velikosti"
- 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 – 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 the add call
 }
 ...
}
```

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 neefektivní, ale metoda nám zajišťuje, že je „scéna" reprezentována polem o velikost odpovídající objektům ve scéně.

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);
}
```

gui

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**.*

Uložení obrázku

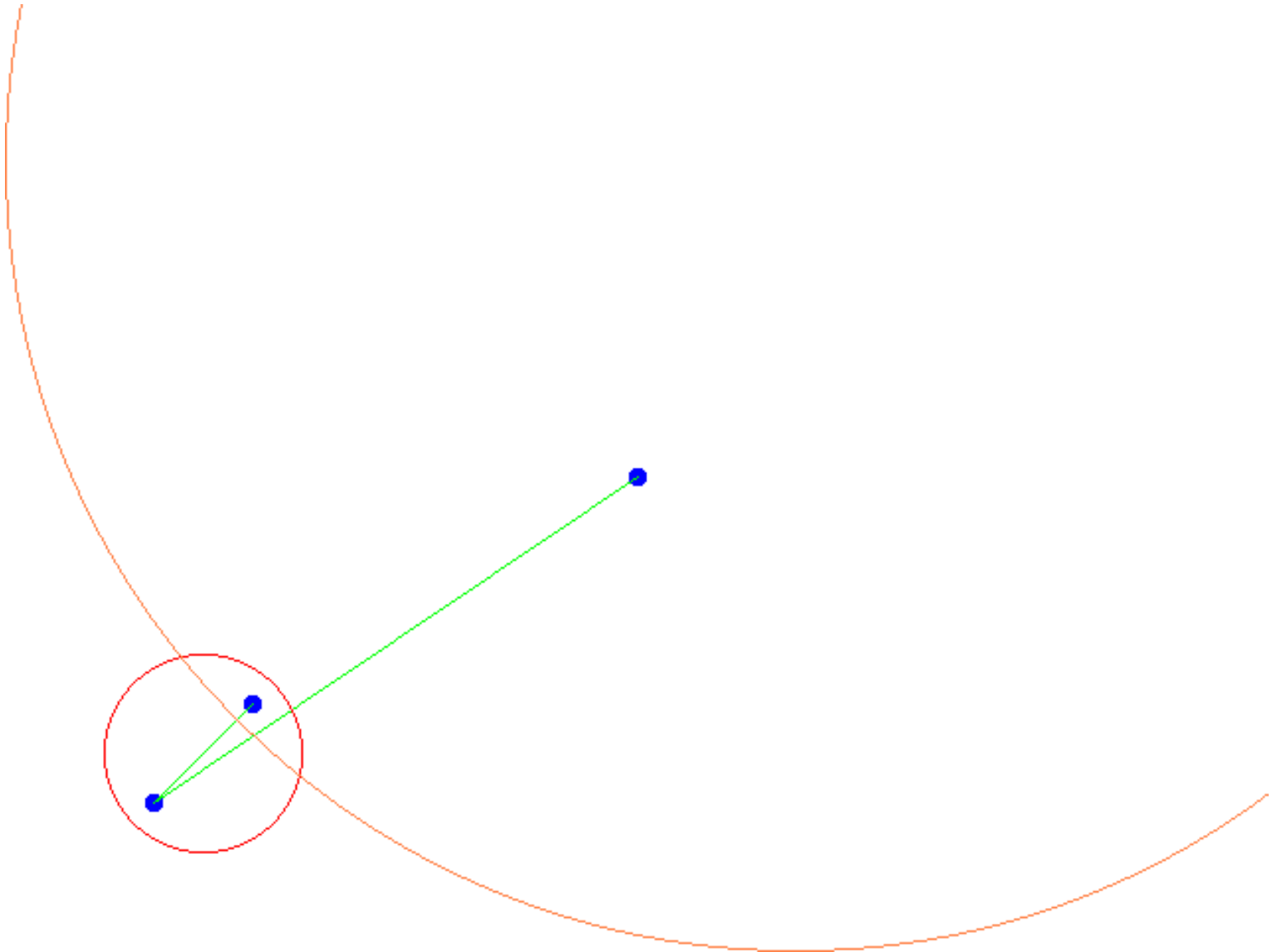
- Uložení plátna můžeme spojit s překreslením aktuální scény v metodě **saveCanvas**

```
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 + "'");
   }
}
```

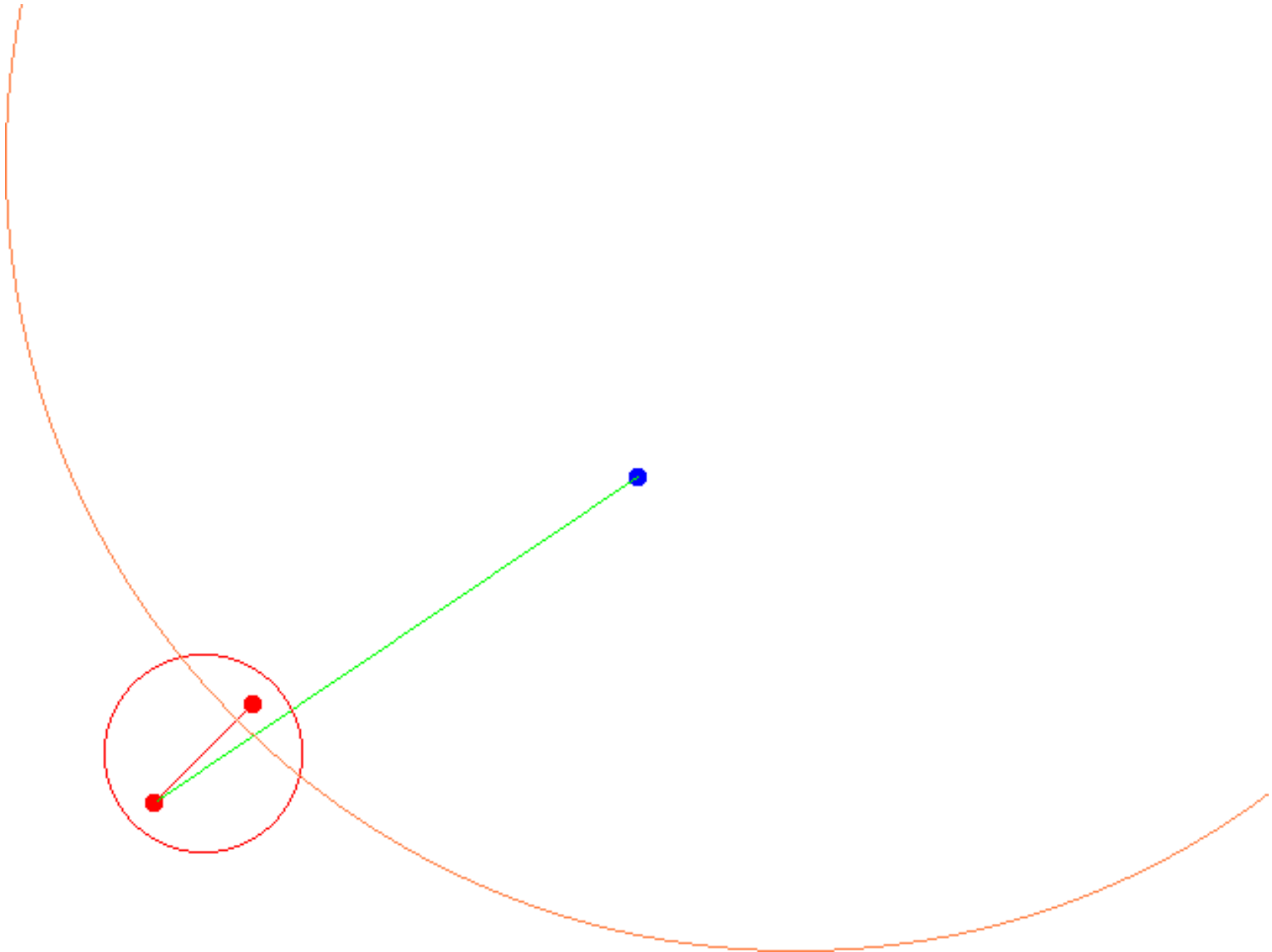
Použití metody **markColorInside**

```
...
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");
```

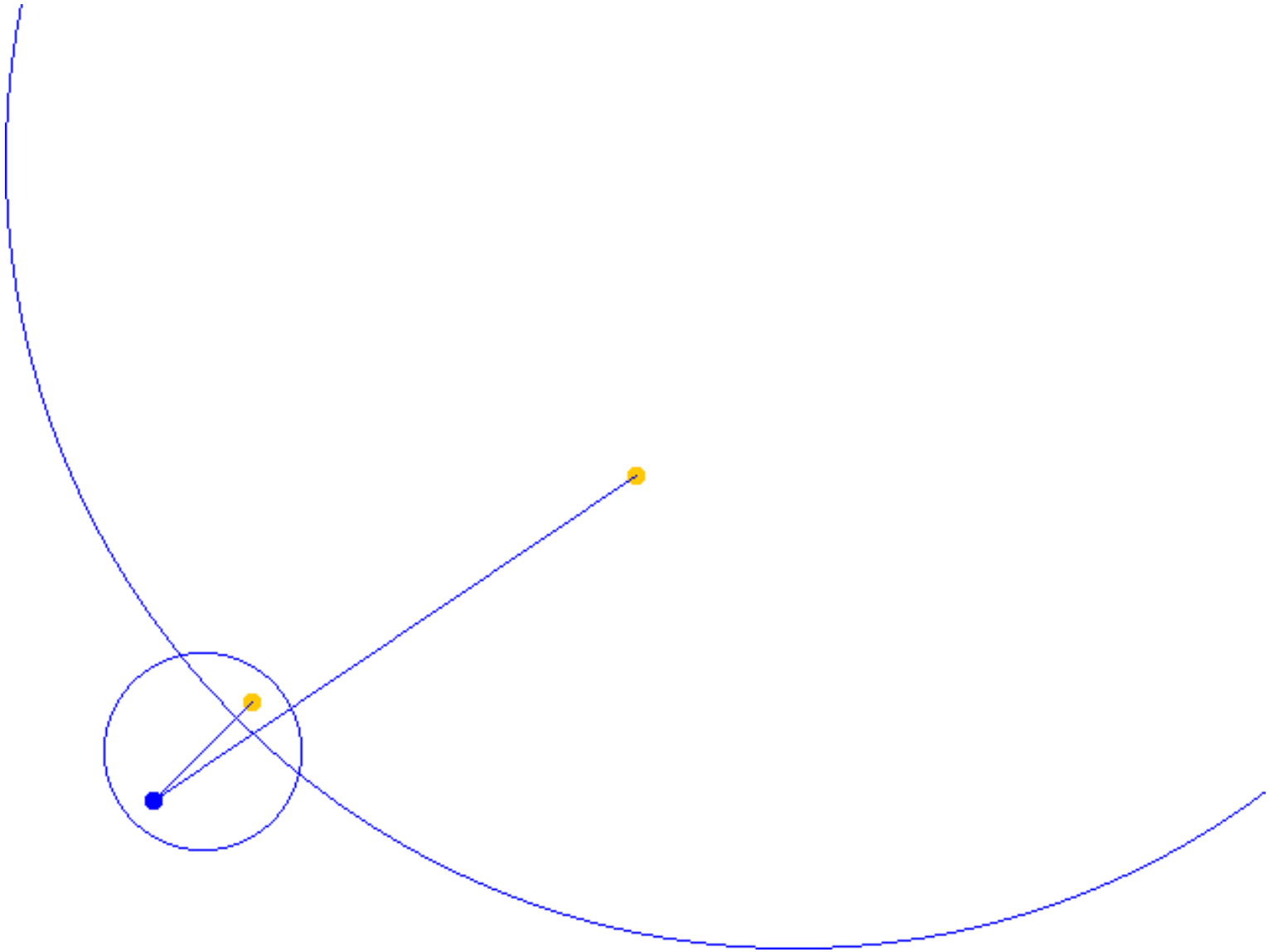
Příklad výstupu – `objects.png`



Příklad výstupu – objects-inside_circle-c1.png



Příklad výstupu – objects-inside_circle-c2.png



Příklad geometrický objektů

- Navrhli jsme hierarchii geometrický objektů
- Implementovali jsme operaci testování, zdali objekt leží uvnitř jiného objektu `isInside`
- Vizualizaci jsem realizovali s využitím rozhraní a implementací z balíku třídy `gui.jar`

- Použití jednotlivých objektů je možné, ale reprezentace scény není pohodlná
- Pro práci s množinou objektů potřebujeme vhodný „kontejner"
- Zatím známe pouze datový typ **pole statické délky**
- Pro efektivní práci s více proměnnými (třeba i jiného typu) potřebujeme složitější **datové struktury**

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