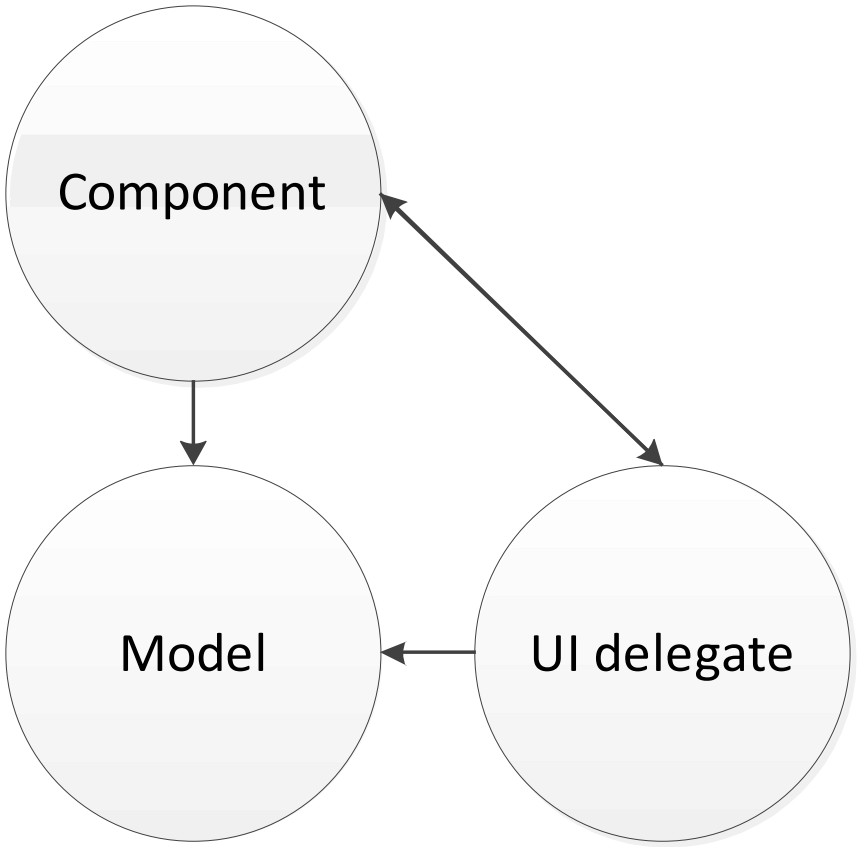
Oprogramowanie Systemów Medycznych

Wykład 4

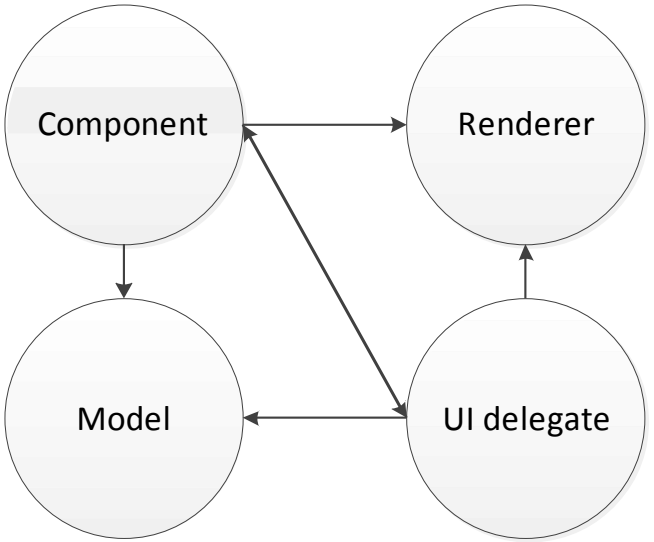
- komponenty Swing wykorzystują architekturę MVC



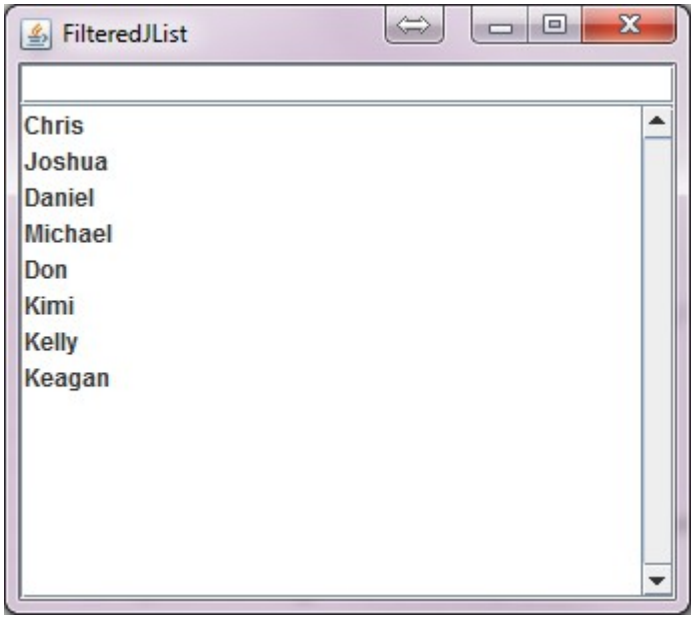
- **Model** - trzymanie danych
- **UI Delegate** - renderowanie danych
- **Component** - komunikacja i koordynacja pomiędzy modelem a delegatem

Dostosowywanie i skalowanie komponentów

- komponenty skalowalne - mogą służyć do obsługi potencjalnie bardzo dużych zbiorów danych: **JTable, JTree, JList, JComboBox**
- Skalowalność realizowana jest przez:
  - Renderer - określa sposób malowania danego typu danych w komórkach (*cell*)
  - Model - dostosowanie modelu danych



Przykład budowania własnego modelu



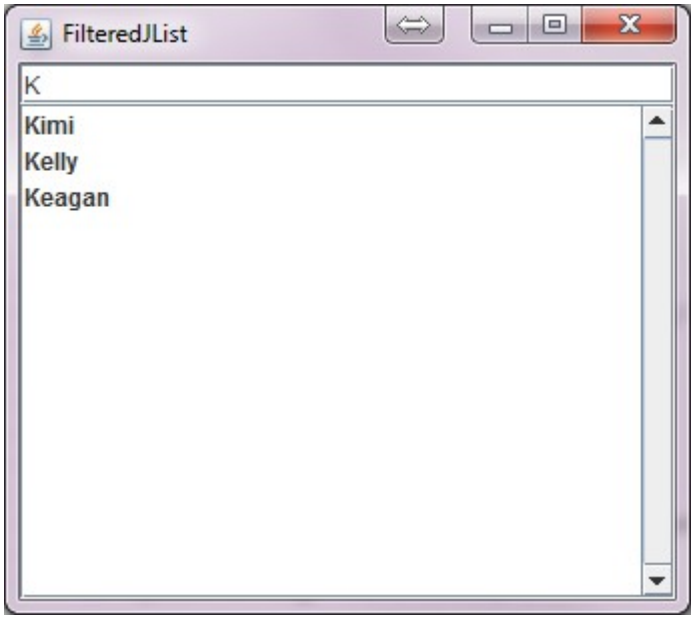
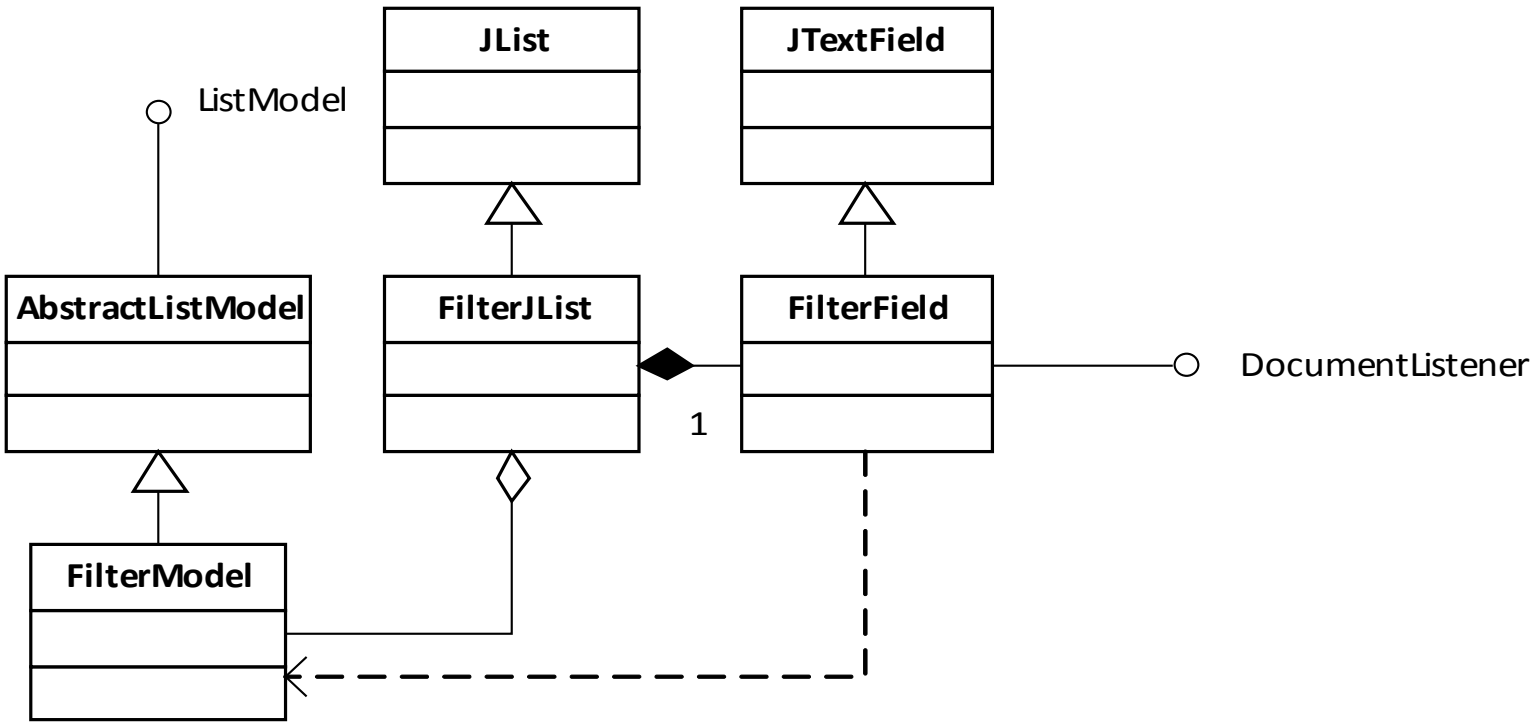


Diagram klas dla przykładu

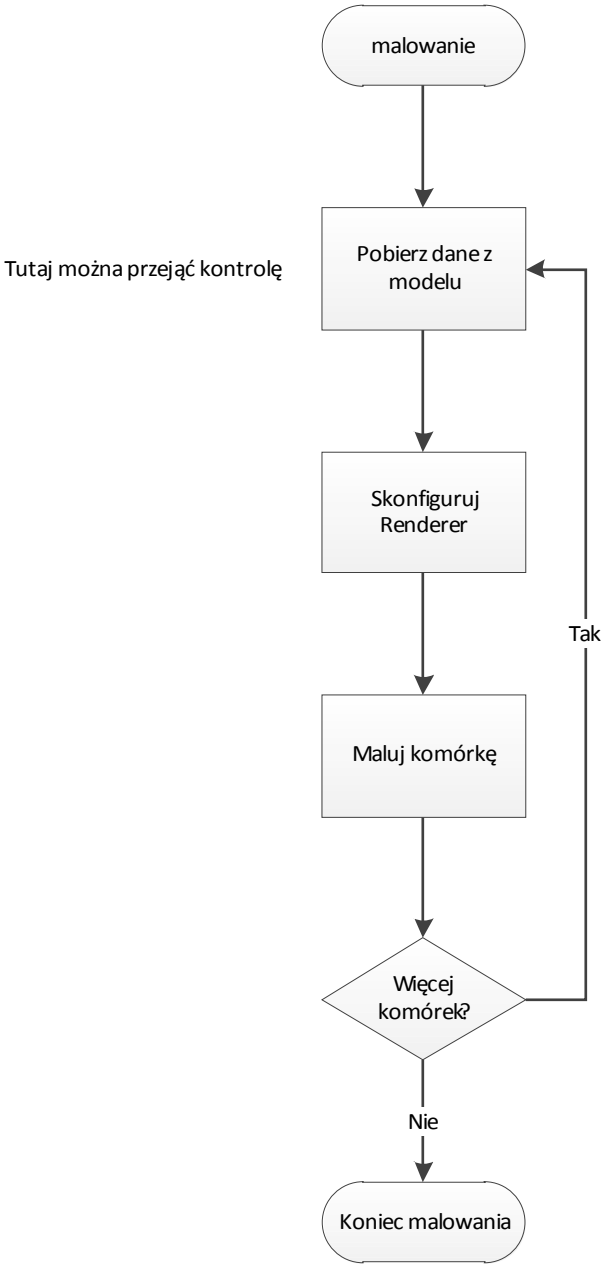


```java
public class FilteredJList extends JList<String> {

    private FilterField filterField;
    private int DEFAULT_FIELD_WIDTH = 20;

    public FilteredJList() {
        super();
        setModel(new FilterModel());
        filterField = new FilterField(DEFAULT_FIELD_WIDTH);

    }

    public void setModel(ListModel<String> m) {
        if (!(m instanceof FilterModel))
            throw new IllegalArgumentException();
        super.setModel(m);
    }

    public void addItem (String o) {
        ((FilterModel) getModel()).addElement (o);
    }

    public JTextField getFilterField() {
        return filterField;
    }
```

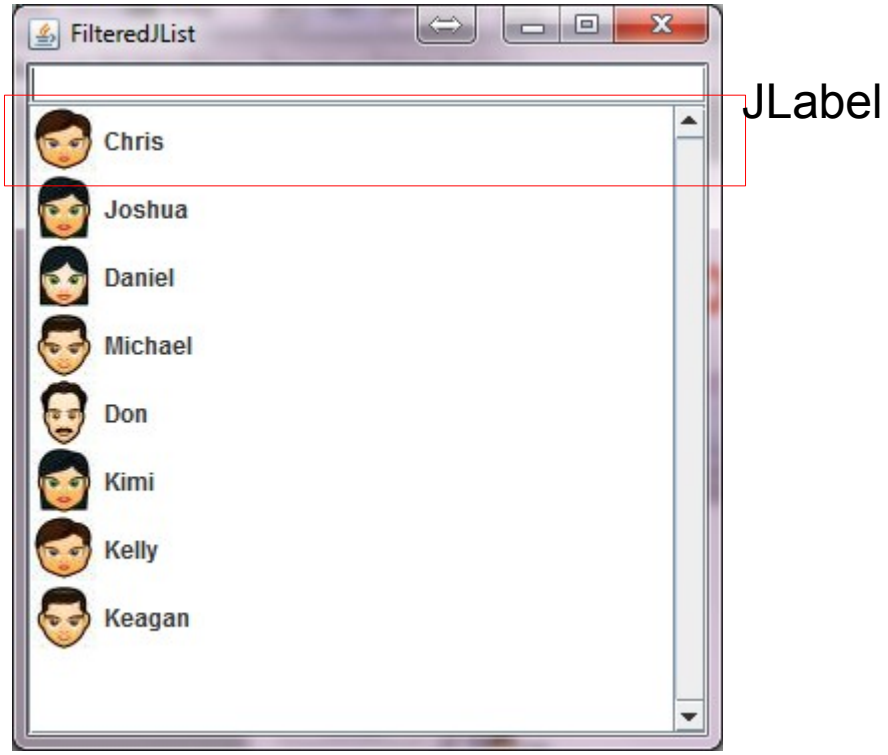
```java
    class FilterField extends JTextField implements
DocumentListener {
        public FilterField(int width) {
            super(width);
            getDocument().addDocumentListener(this);
        }

        public void changedUpdate(DocumentEvent e) {
            ((FilterModel) getModel()).refilter();
        }

        public void insertUpdate(DocumentEvent e) {
            ((FilterModel) getModel()).refilter();
        }

        public void removeUpdate(DocumentEvent e) {
            ((FilterModel) getModel()).refilter();
        }
    }
```

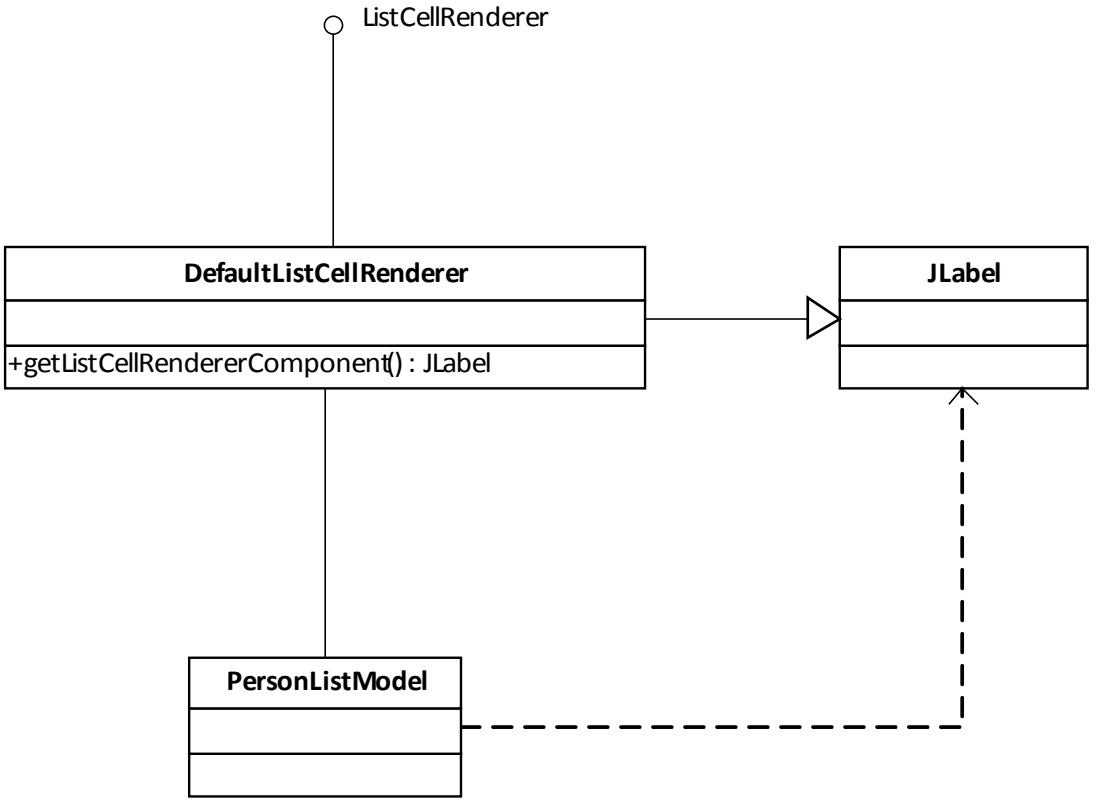
```java
    class FilterModel extends AbstractListModel<String> {
        ArrayList<String> items;
        ArrayList<String> filterItems;

        public FilterModel() {
            super();
            items = new ArrayList<>();
            filterItems = new ArrayList<>();
        }

        public String getElementAt(int index) {
            if (index < filterItems.size())
                return filterItems.get(index);
            else
                return null;
        }

        public int getSize() {
            return filterItems.size();
        }

        public void addElement(String o) {
            items.add(o);
            refilter();

        }

        private void refilter() {
            filterItems.clear();
            String term = getFilterField().getText();
            for (int i = 0; i < items.size(); i++) {
                if (items.get(i).toString().startsWith(term)) {
                    filterItems.add(items.get(i));
                }
            }
            fireContentsChanged(this, 0, getSize());
        }
    }
```

Renderowanie komponentu skalowalnego



Tworzenie własnych Renderer'ów





```
class PersonCellRenderer extends DefaultListCellRenderer {

    private Map<String, String> icons;
    public PersonCellRenderer( Map<String, String> icons ) {
        this.icons = icons;
    }
    @Override
    public Component getListCellRendererComponent(JList<?> list,
            Object value, int index, boolean isSelected,
            boolean cellHasFocus) {
        JLabel cell = (JLabel) super.getListCellRendererComponent(list, value, index, isSelected,
                cellHasFocus);

        ImageIcon icon = null;
        String itemText = (String) value;
        if ( icons.containsKey( itemText ) ){
            try {
                BufferedImage img = ImageIO.read( FilteredJList.class.getResource( icons.get(itemText) ) );
                icon = new ImageIcon( img );
            } catch (IOException e) {
                // ignore
            }
            if ( icon != null ) {
                cell.setIcon( icon );
            }
        }
        return cell;
    }
}
```

Dalsze modyfikacje wyglądu i zachowania

- zmiana Look and Feel (LaF) za pomocą UIManager
- zmiana wartości domyślnych dla danego LaF

```
UIDefaults def = UIManager.getLookAndFeelDefaults();
Enumeration<?> enu = def.keys();
while (enu.hasMoreElements()) {
        Object item = enu.nextElement();
        System.out.println(item +" " + def.get(item));
}
```

```
RadioButtonMenuItem.foreground sun.swing.PrintColorUIResource[r=51,g=51,b=51]
```

- stworzenie własnego delegata UI, poprzez rozszerzenie ComponentUI, lub klas potomnych (np, ButtonUI), a następnie instalacja nowego delegata
- stworzenie własnego komponentu i przeładowanie metody **JComponent.paintComponent(Graphics g)**

Renderowanie komponentów

przestrzeń użytkownika

przestrzeń urządzenia

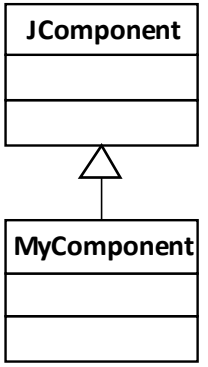
Malowanie własnego komponentu

- **`paint -> paintComponent, paintBorder, paintChildren`**
  - jest wywoływana przez Swing kiedy komponent ma być namalowany na ekranie
- **`paint`** jest często wywoływane w odpowiedzi na zdarzenia
- **`repaint`** woła **`update`**, która wymusza wykonanie **`paint`** w *możliwie niedalekiej przyszłości*

```
public void repaint()
public void update( Graphics g )
public void paint( Graphics g )
public void paintComponent( Graphics g )
```

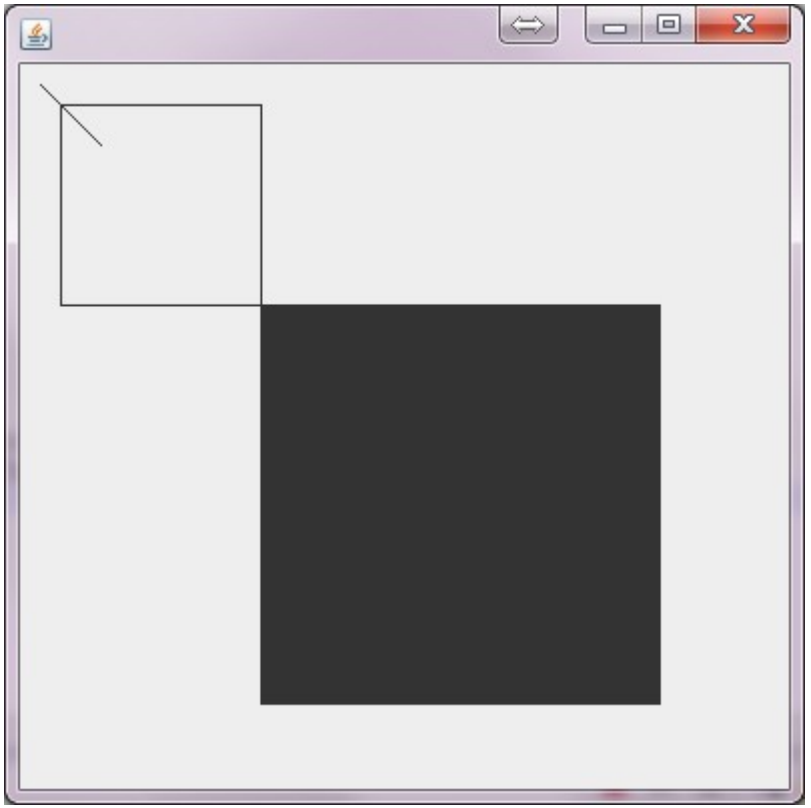
Malowanie własnego komponentu

```
class MyComponent extends JComponent {

    @Override
    protected void printComponent(Graphics g) {
        Graphics2D g2 = (Graphics2D) g;

        // malowanie
    }

}
```



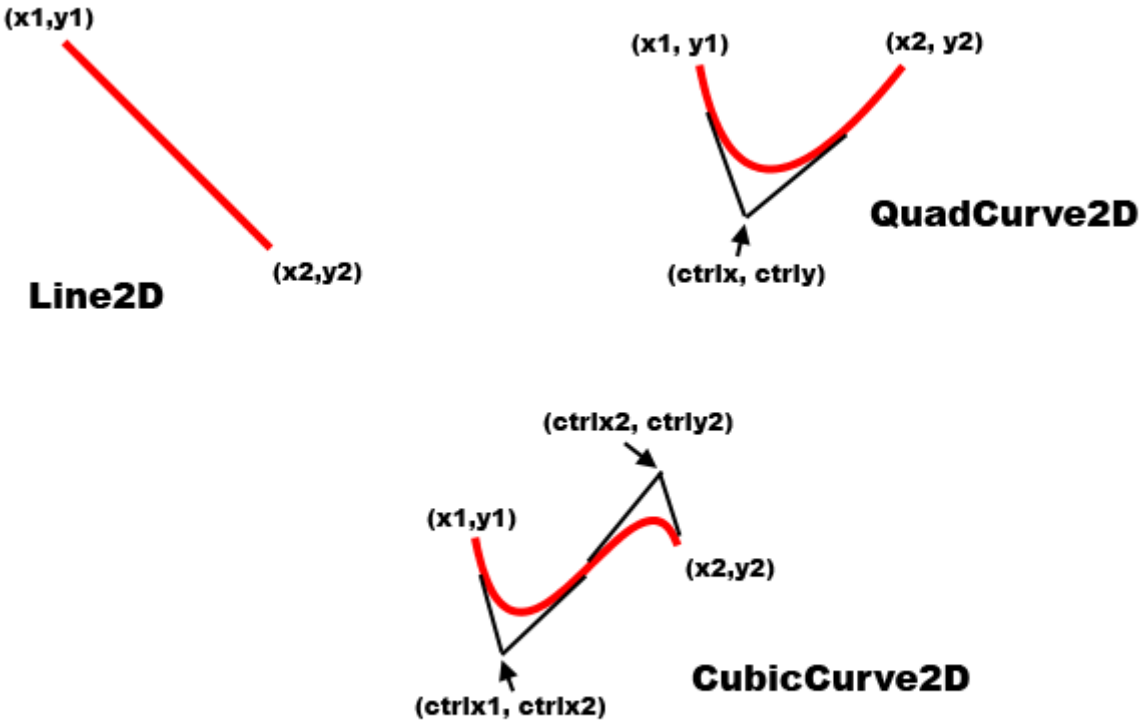
Malowanie prostych kształtów

```
public void paintComponent( Graphics g ) {
    g.drawLine(10, 10, 40, 40);
    g.drawRect(20, 20, 100, 100);
    g.fillRect(120, 120, 200, 200);
}
```

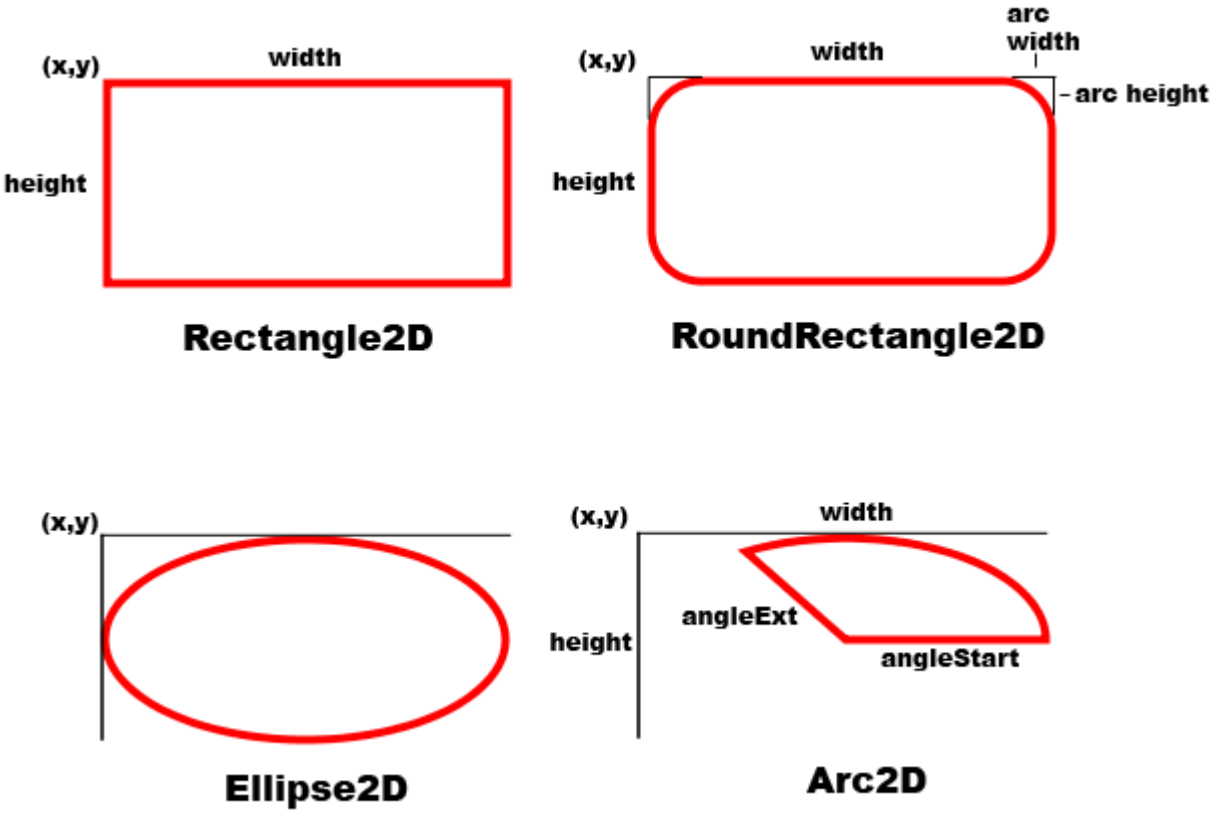


Linie i krzywe

- **`java.awt.geom.Line2D`** - linia prosta
- **`java.awt.geom.QuadCurve2D`** - krzywa z jedym punktem kontrolnym
- **`java.awt.geom.CubicCurve2D`** - krzywa z dwoma punktami kontrolnymi

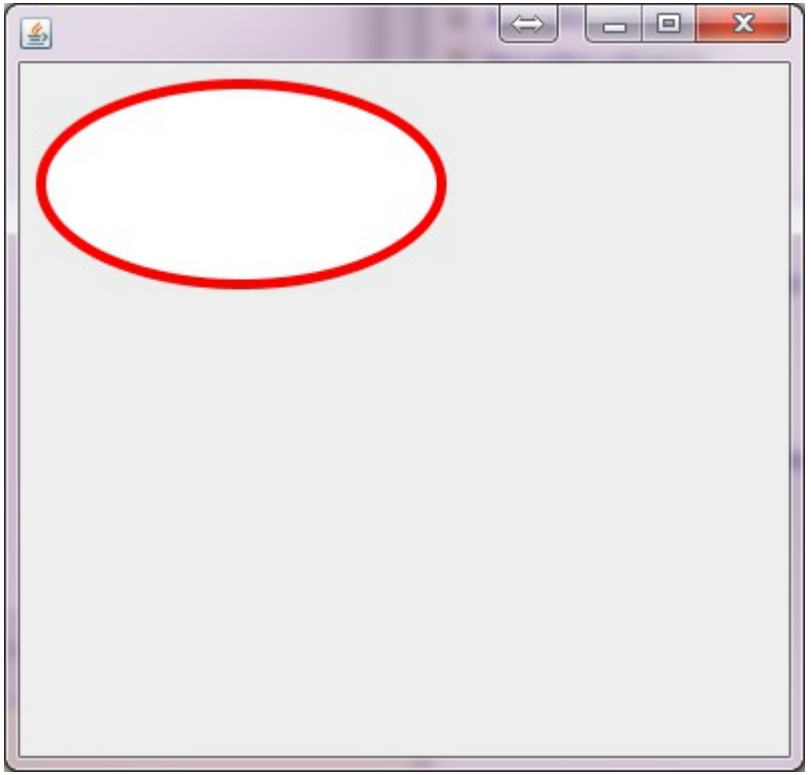


Rysowanie prostokątów, elips i łuków



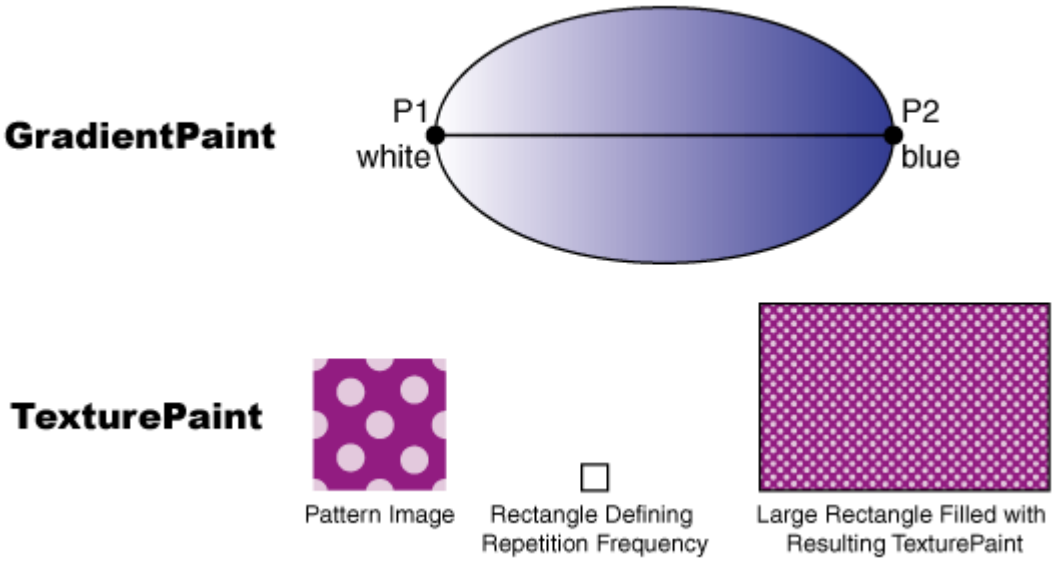
Elipsa

```
        public void paintComponent( Graphics g ) {
            Graphics2D g2d = (Graphics2D) g;
            Ellipse2D myEllipse = new Ellipse2D.Double(10.0, 10.0, 200.0,
100.0);
            g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
            RenderingHints.VALUE_ANTIALIAS_ON);
            g2d.setStroke(new BasicStroke(5));
            g2d.setPaint(Color.white);
            g2d.fill(myEllipse);
            g2d.setPaint(Color.red);
            g2d.draw(myEllipse);
        }
```



Farby (paint)

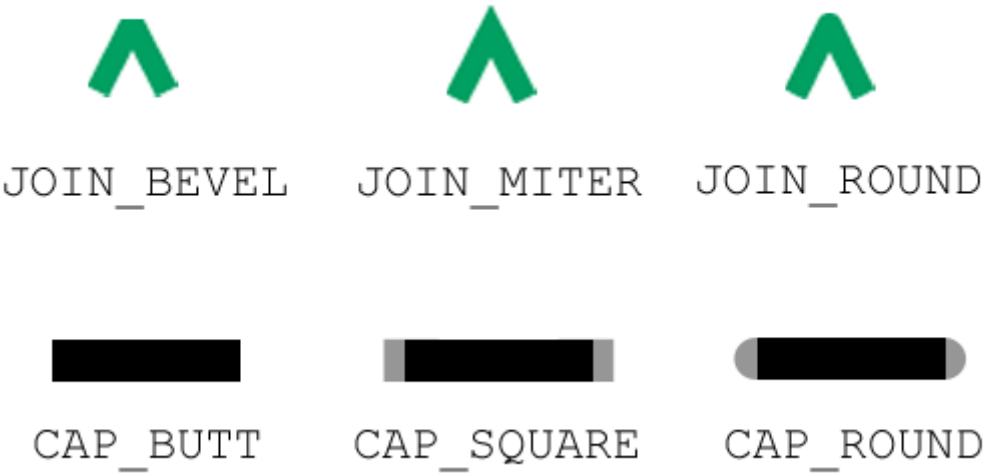
- wypełnienie kształtów zapewniają klasy implementujące interfejs **`java.awt.Paint`**
  - **`java.awt.Color`** - wypełnienie kolorem
  - **`java.awt.GradientPaint`** - wypełnienie gradientem
  - **`java.awt.TexturePaint`** - wypełnienie teksturą



Pędzle (stroke)

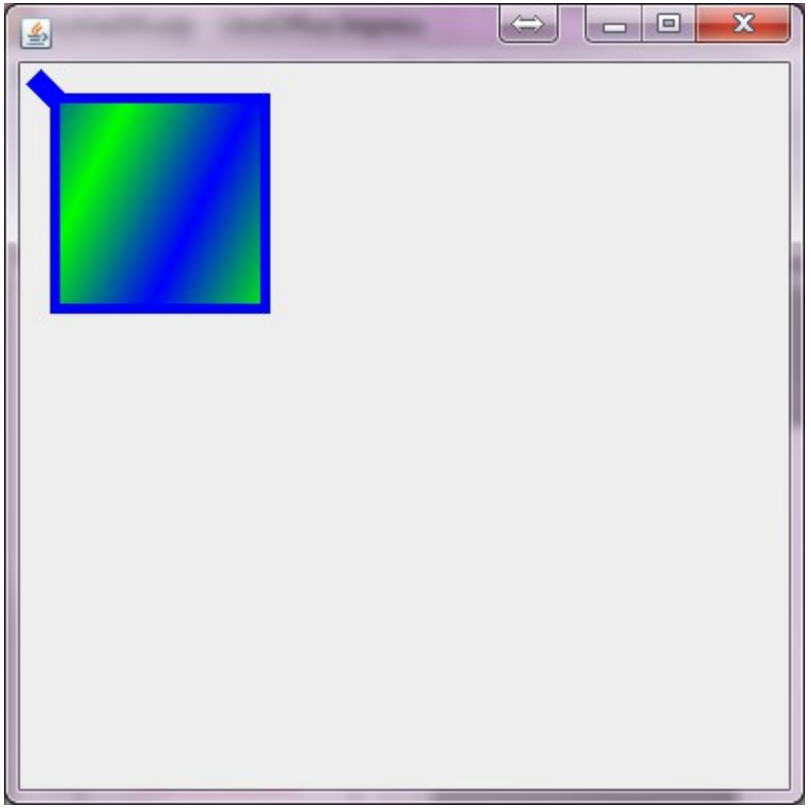
- java.awt.Stroke - interfejs określający styl malowania konturów
  - grubość
  - rodzaj linii
  - rodzaj połączeń
- java.awt.BasicStroke()

```
Stroke stroke = new BasicStroke(5.0f , // szerokość
                    BasicStroke.CAP_ROUND,  // styl końca lini
                    BasicStroke.JOIN_MITER, // styl połączenia
                    15.0f,                  // miter
                    new float[] {10.0f,10.0f}, // kreskowanie
                    5.0f);                  // faza kreskowania

g2d.setStroke( stroke );
```



Kolor i wypełnienie

```
        public void paintComponent( Graphics g ) {
            Graphics2D g2 = (Graphics2D)g;
            Line2D line = new Line2D.Double(10, 10, 40, 40);
            g2.setColor(Color.blue);
            g2.setStroke(new BasicStroke(10));
            g2.draw(line);
            Rectangle2D rect = new Rectangle2D.Double(20, 20, 100, 100);
            g2.draw(rect);
            g2.setPaint(
            new GradientPaint(0, 0, Color.blue, 50, 25, Color.green, true));
            g2.fill(rect);
        }
```



Malowanie tekstu

- **`java.awt.Font`** - czcionka tekstu

```
public Font( String name, int style, int size)
```

  - name: nazwa fontu dostępnego w systemie (np. Arial, Times New Roman)
  - styl fontu: **`Font.PLAIN`**, **`Font.ITALIC`**, **`Font.BOLD`**
    - można tak: **`Font.BOLD | Font.ITALIC`**

```
Graphics2D g2d = (Graphics2D) g;
Font font = new Font("Comic Sans MS", Font.BOLD | Font.ITALIC, 20 );
g2d.setFont(font);
g2d.setRenderingHint( RenderingHints.KEY_ANTIALIASING,
RenderingHints.VALUE_ANTIALIAS_ON );
g2d.drawString("Hello world!", 20, 20 );
```

Metryka fontu

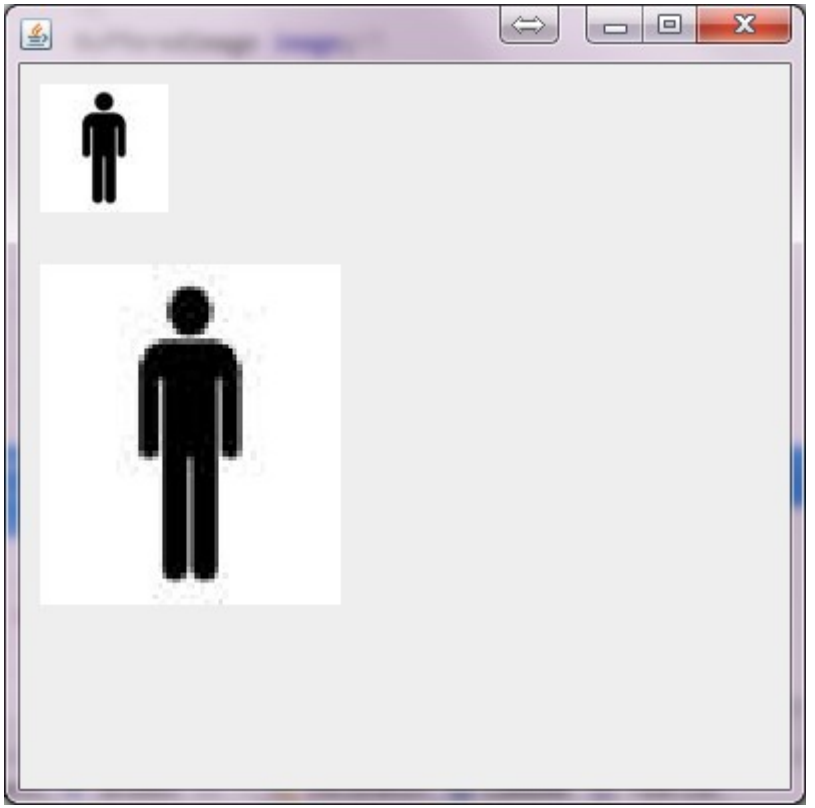
- **`java.awt.FontMetrics`** - służy do określania rozmiarów czcionki - użyteczna do pozycjonowania tekstu

```
FontMetrics fontMerics = g2d.getFontMetrics();
```

Malowanie obrazów

```
try {
	image = ImageIO.read( new File("image.jpg") );
} catch (IOException e) {
	// hadle it
}
```

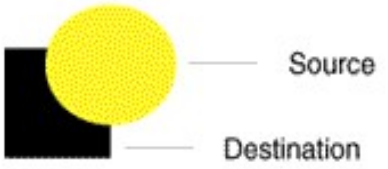
```
public void paintComponent( Graphics g ) {
    Graphics2D g2d = (Graphics2D) g;
    g2d.drawImage(image, 10, 10, null );
    g2d.drawImage(image, 10, 100, 150, 170, null );
}
```



Kompozycja

- reguły ustalania kolejności rysowania kształtów (compositing)

```
g2d.setComposite( AlphaComposite.getInstance( AlphaComposite.SRC_OVER) );
```



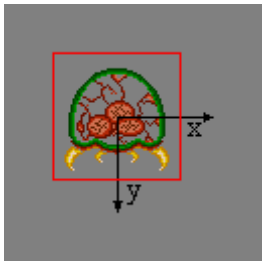
Transformacje

- transformacja afiniczna - opisane macierzą przekształcenie układu współrzędnych realizujące **przesunięcie**, **obrót**, **skalowanie** i **przekoszenie.**

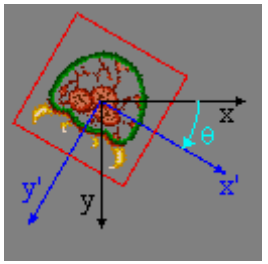
$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} m_{01}x + m_{01}y + m_{02} \\ m_{10}x + m_{11}y + m_{12} \\ 1 \end{bmatrix}$$

Transformacje afiniczne

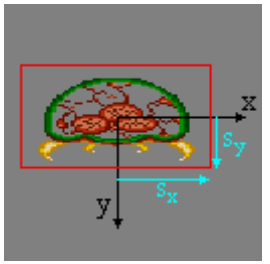
Identity



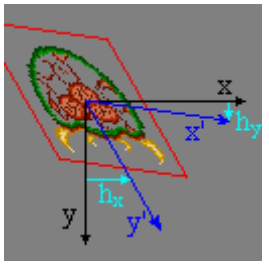
Obrót



Skalowanie



Przekoszenie



$$I=\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{bmatrix}$$

$$R=\begin{bmatrix}\cos(\theta) & \sin(\theta) & 0\\ -\sin(\theta) & \cos(\theta) & 0\\ 0 & 0 & 1\end{bmatrix}$$

$$S=\begin{bmatrix}s_x & 0 & 0\\ 0 & s_y & 0\\ 0 & 0 & 1\end{bmatrix}$$

$$A=\begin{bmatrix}1 & k & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{bmatrix}$$

Przesunięcie

$$T=\begin{bmatrix}1 & 0 & t_x\\ 0 & 1 & t_y\\ 0 & 0 & 1\end{bmatrix}$$

Każda przekształcenie posiada przekształcenie odwrotne

Kolejność

- przekształcenia można składać mnożąc przez siebie macierze przekształceń
- UWAGA: kolejność ma znaczenie!
  1. rotacja
  2. skalowanie
  3. przesunięcie

$$P = R_\theta \cdot S \cdot T = \begin{bmatrix} s_x \cos(\theta) & s_y \sin(\theta) & t_x \\ -s_x \sin(\theta) & s_y \cos(\theta) & t_y \\ 0 & 0 & 1 \end{bmatrix}$$

- np. obrót wokół punktu
  1. przesuń oś obrotu do punktu
  2. obróć
  3. przesuń odwrotnie

$$P = T \cdot R_\theta \cdot T^{-1}$$

AffineTransform

- **`java.awt.geom.AffineTransform`** - macierz przekształcenia

```
@Override
protected void paintComponent(Graphics g )
{
    double x = getWidth()/2;
    double y = getHeight()/2;

    Graphics2D g2d = (Graphics2D) g;
    Rectangle2D rect = new Rectangle2D.Double(0, 0, 20, 30);

    AffineTransform t = new AffineTransform();
    t.setToIdentity();
    t.translate( x-10, y-15 );
    g2d.draw( t.createTransformedShape(rect) );

    g2d.setPaint( Color.RED );
    t.setToTranslation( x-10, y-15 );
    t.translate( 10, 15 );
    t.rotate( Math.PI * 0.3 );
    t.scale(4.0, 1.0);
    t.translate( -10, -15 );
    g2d.draw( t.createTransformedShape(rect) );
}
```