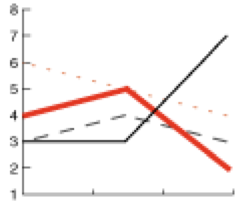
Advanced Programming
Java 2D Graphics

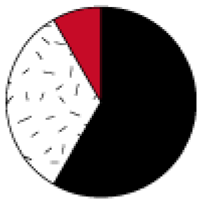
Computer Graphics

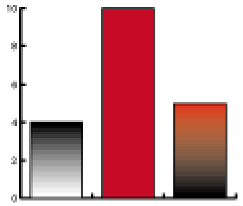
- **Computer Graphics**: the *representation* and *management* of visual content: drawings, charts, photographs, movies, etc.
- 2D, 3D, Raster (Pixel), Vector, Animation, etc.
- **Rendering**: generating an image from a model using a computer, defining its *shape, color, texture, transparency, shades, etc.*
- Support for different types of devices: screen, memory, printer, plotter, etc.
- User Space → Device Space

Java 2D

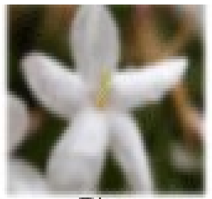
- **Two-dimensional graphics, text, and imaging**
- A **uniform rendering model** for display devices and printers
- **Geometric primitives**: any geometric shape
- **Hit detection** on shapes, text, and images
- Ccontrol over how **overlapping** objects are rendered
- Enhanced **color support** that facilitates color management
- Support for **printing** complex documents
- Control of the **quality** of the rendering (hints)







Image


Blur

Sharpen

The “Drawing” Concept

- Graphical interfaces are built using **components**.

  The “system” draws the components automatically:
  - when they are displayed for the first time,
  - at minimize, maximize operations,
  - when resizing the display area;
- The **support methods** for defining the graphical representation of a *Component* are:
  - **void paint(Graphics g)**
  - **void update(Graphics g)**
  - **void repaint()**

The *paint* method

This method is called when the contents of the component should be painted; such as when the component is first being shown or is damaged and in need of repair. The *clip rectangle* in the *Graphics* parameter is set to the area which needs to be painted.

```
public class MyFrame extends Frame {
  public MyFrame(String title) {
    super(title);
    setSize(200, 100);
  }

  public void paint(Graphics g) {
    super.paint(g);
    // Apelam metoda paint a clasei Frame
    g.setFont(new Font("Arial", Font.BOLD, 11));
    g.setColor(Color.red);
    g.drawString("DEMO Version", 5, 35);
  }
}
```

The *paintComponent* method

- *JComponent.paint* delegates the work of painting to three protected methods: **paintComponent**, **paintBorder**, and **paintChildren**. They're called in the order listed to ensure that children appear on top of component itself.
- Swing components should just override paintComponent.

```
/* Creating a custom component */
class MyCustomComponent extends JPanel {

    // Define the representation of the component
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        Graphics2D g2d = (Graphics2D) g;
        ...
    }
    // Methods used by the layout managers
    public Dimension getPreferredSize() { return ... };
    public Dimension getMinimumSize() { return ... }
    public Dimension getMaximumSize() { return ... }
}
```

Creating a Custom Component

```
public class MyComponent extends JPanel {
    private int x, y, radius;
    public MyComponent() {
      init();
    }
    private void init() {
      setPreferredSize(new Dimension(400, 400));
      this.addMouseListener(new MouseAdapter() {
          public void mousePressed(MouseEvent e) {
              x = e.getX(); y = e.getY();
              radius = 50 + (int) (100 * Math.random());
              repaint();
          }
      });
    @Override
    public void paintComponent(Graphics g) {
      super.paintComponent(g);
      g.drawOval(x - radius / 2, y - radius / 2, radius, radius);
    }
}
```

```
JFrame frame = new JFrame("demo");
frame.add(new MyComponent());
frame.pack();
frame.setVisible(true);
```

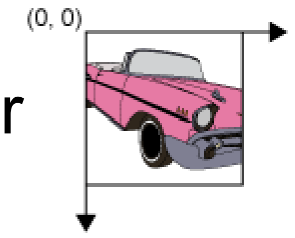
Graphics, Graphics2D

- ***Graphics*** is the base class for all **graphics contexts** that allow an application to draw onto components realized on various devices, as well as onto off-screen images.
- ***Graphics2D*** class extends the *Graphics* class to provide more sophisticated control over geometry, coordinate transformations, color management, and text layout.
- A graphic context offers:
  - Methods for configuring the **drawing properties**:

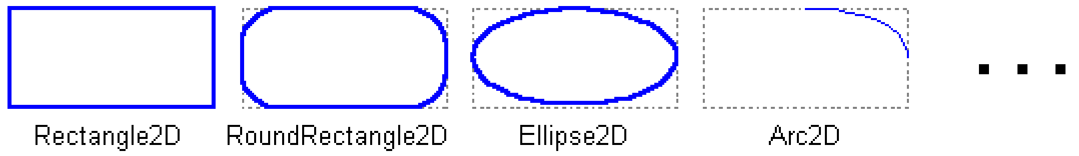
    color, paintMode, font, stroke, clip, renderingHints, ...
  - **Geometric primitives**
  - Support for working with **texts** and **images**
  - Support for **printing**

Geometric Primitives

- **Coordinates**
  - **User space** – in which graphics primitives are specified
  - Device space – screen, window, or a printer
  - The origin of user space is the upper-left corner



- **Primitives**:
  - `drawLine, drawPolyline, drawOval, fillOval, drawPolygon, fillPolygon, drawRect, fillRect, …`
  - **`draw(Shape), fill(Shape)`**
  - The *Shape interface* provides definitions for objects that represent some form of geometric shape. The Shape is described by a PathIterator object, which can express the outline of the Shape as well as a rule for determining how the outline divides the 2D plane into interior and exterior points.



Working with Texts

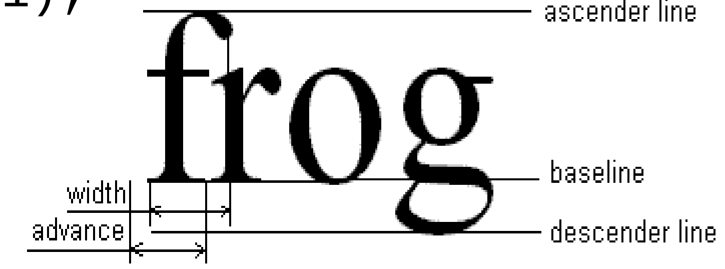
- **Font** - A collection of *glyphs* (unique marks that collectively add up to the spelling of a word) → *name*, *style*, *size*

```
Label label = new Label("Some text");
label.setFont(new Font("Dialog", Font.PLAIN, 12));

void paint(Graphics g) {
    g.setFont(new Font("Courier", Font.BOLD, 10));
    g.drawString("Another text", 10, 20); }
```

- **FontMetrics** - encapsulates information about the rendering of a particular font on a particular screen.

```
Font f = new Font("Arial", Font.BOLD, 11);
FontMetrics fm = g.getFontMetrics();
int height = fm.getHeight();
int width = fm.stringWidth("frog");
int xWidth = fm.charWidth('g');
```



- **TextLayout** - highlighting, strings with mixed fonts, mixed languages, bidirectional text.

Using Colors

- **Paint interface** defines how color patterns can be generated for Graphics2D operations.
- **Color** encapsulates colors in the sRGB space

Red Green Blue Alpha
(0 – 255, 0.0 – 1.0)

```
Color standardRed = Color.RED;
Color plainWhite = new Color(1.0, 1.0, 1.0);
Color translucentRed = new Color(255, 0, 0, 128);
```

- **SystemColor** encapsulate symbolic colors representing the color of native GUI objects on a system.

```
SystemColor.desktop
```

- **GradientColor** provides a way to fill a *Shape* with a linear color gradient pattern.

Hello world!

- **TexturePaint** provides a way to fill a *Shape* with a texture that is specified as a *BufferedImage*.

Hello again...

Using Images

- **Image** is the superclass of all classes that represent graphical images.

- **BufferedImage**
  - Loadind from a file

    ```
    BufferedImage image = ImageIO.read(new File("hello.jpg"))
    ```

  - Creating in memory (off-screen)

    ```
    BufferedImage image = new BufferedImage(w, h, type);
    Graphics g = image.getGraphics();
    ```

  - Drawing using a graphic context

    ```
    graphics.drawImage(image);
    ```

  - Saving in a file (GIF, PNG, JPEG, etc.)

    ```
    ImageIO.write(image, "png", new File("drawing.png"));
    ```

Working with Large Images

- Displaying a large image

```
BufferedImage img = ImageIO.read(
    new URL("http://www.remoteServer.com/hugeImage.jpg"));
...
public void paint(Graphics g) {
    g.drawImage(img, 0, 0, this);
}
```

- *ImageObserver* **-** an asynchronous update interface for receiving notifications about information as the *Image* is constructed.

```
public boolean imageUpdate(Image image, int flags, int x, int y,
            int width, int height) {
  // If the image has finished loading, repaint the window.
  if ((flags & ALLBITS) != 0) {
     repaint();
     return false;  // finished, no further notification.
  }
  return true; //not finished loading, need further notification.
}
```

Double-Buffering

Create an offscreen image, draw to that image using the image's graphics object, then, in one step, call *drawImage* using the target window's graphics object and the offscreen image. Swing uses this technique by default.

Preventing *flickering*

```
// Override update, we don't need it anymore
public void update(Graphics g) {
  paint(g);
}
public void paint(Graphics g) {
  BufferedImage offImage =
    new BufferedImage(100, 200, BufferedImage.TYPE_INT_ARGB);
  Graphics2D g2 = offImage.getGraphics();
  // Draw off-screen
  g2.setColor(...);
  g2.fillOval(...); ...
  // Transfer the drawing: back buffer -> primary surface (screen)
  g.drawImage(offImage, 0, 0, this);
  g2.dispose();
}
```

Printing

- Creat a component that implements *Printable* interface

```
public class HelloWorldPrinter implements Printable {
    public int print(Graphics g, PageFormat pf, int page)
        throws PrinterException {
        if (page > 0) {
            return NO_SUCH_PAGE;
        }
        g.drawString("Hello world!", 100, 100);
        return PAGE_EXISTS;
    }
}
```

- Create a *PrinterJob*

```
PrinterJob job = PrinterJob.getPrinterJob();
job.setPrintable(new HelloWorldPrinter());
if (job.printDialog()) {
   job.print();

}
```

The *Printable.print()* method is called by the printing system, just as the *Component.paint()*

- Some Swing components are printing-aware *(JTable, JTextComponent)*

Java Tutorial

- **Trail: 2D Graphics**

  http://docs.oracle.com/javase/tutorial/2d/index.html
- **Lesson: Full-Screen Exclusive Mode API**

  http://docs.oracle.com/javase/tutorial/extra/fullscreen/index.html
- **Trail: Sound**

  http://docs.oracle.com/javase/tutorial/sound/index.html
- **Java Demos → Java2D application**