# Simple Java Graphics

Originals of slides and source code for examples: http://courses.coreservlets.com/Course-Materials/java.html
Also see Java 8 tutorial: http://www.coreservlets.com/java-8-tutorial/ and many other Java EE tutorials: http://www.coreservlets.com/
Customized Java training courses (onsite or at public venues): http://courses.coreservlets.com/java-training.html



For additional materials, please see http://www.coreservlets.com/. The Java tutorial section contains complete source code for all examples in this tutorial series, plus exercises and exercise solutions for each topic.

**coreservlets.com** – custom onsite training




## For customized training related to Java or JavaScript, please email hall@coreservlets.com

Marty is also available for consulting and development support

Taught by lead author of *Core Servlets & JSP*, co-author of *Core JSF* (4th Ed), and this tutorial.

Available at public venues, or custom versions can be held on-site at your organization.

- **Courses developed and taught by Marty Hall**
  - JSF 2.3, PrimeFaces, Java programming (using Java 8, for those new to Java), Java 8 (for Java 7 programmers), JavaScript, jQuery, Ext JS, Spring Framework, Spring MVC, Java EE 8 MVC, Android, GWT, custom mix of topics
  - Courses available in any state or country.
  - Maryland/DC companies can also choose afternoon/evening courses.
- **Courses developed and taught by coreservlets.com experts (edited by Marty)**
  - Hadoop, Hibernate/JPA, HTML5, RESTful Web Services

**Contact hall@coreservlets.com for details**

## Topics in This Section

- Simple popup windows *using* JFrame
- Variation *extending* JFrame
- Drawing from paintComponent
- Lifecycle methods and value of @Override
- Drawing images and automatic threads
- Handling exceptions with try/catch blocks





# Simple Popup Windows



## Overview

- **JFrame**
  - Simple popup window
    - Lets us perform simple drawing operations
    - Illustrates lifecycle methods, overriding, try/catch blocks
- **Basic usage**
  - Instantiate the JFrame and assign text for the title bar

```
JFrame frame = new JFrame("Title");
```

  - Specify size

```
frame.setSize(width, height);
```

  - Designate that closing the window ends the program

```
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

  - Pop up the window

```
frame.setVisible(true);
```



## Variations on the Theme

- **First variation**
  - *Using* a JFrame
    - Simplest usage, with main method having variable of type JFrame
- **Second variation**
  - *Extending* a JFrame
    - Better usage, with class that extends JFrame, so that the class be used multiple places
- **Later variations**
  - Replacing the content pane of a JFrame
    - Assign a custom subclass of JPanel as the content, so that you can:
      - Override paintComponent for drawing
      - Assign event handlers to respond to user actions

# Using a JFrame

```
package coreservlets;

import javax.swing.*;

public class Popup1 {
  public static void main(String[] args) {
    JFrame frame = new JFrame("Simple Popup Window");
    frame.setSize(500,500);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setVisible(true);
  }
}
```

Use packages to avoid class name conflicts.

No need to memorize the packages (folders that contain the classes you are using). Instead, just start typing the code into Eclipse. When Eclipse sees unknown class, it will put red X on the line, but if you click on the lightbulb next to the X (or hit Control-1), Eclipse will offer to insert the imports for you.

For brevity, I will omit package statements and imports in most code listings.

These are the four operations shown on the Overview slide. However, since I perform these operations directly in main, it is difficult to reuse this code in other places.

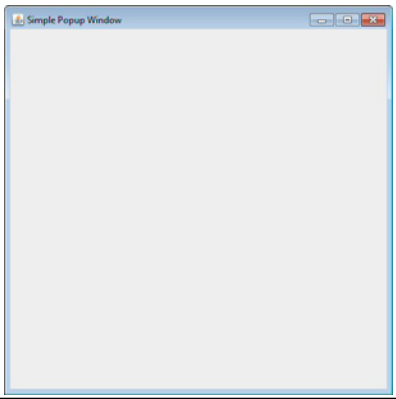


# Extending a JFrame

```
public class Popup2 extends JFrame {
  public Popup2() {
    super("Simple Popup Window");
    setSize(500,500);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setVisible(true);
  }

  public static void main(String[] args) {
    new Popup2();
  }
}
```

Since I extend JFrame, other methods can easily instantiate my class.

I put main in the same class for testing, but I could put the same code in main in another class, or in any other method.







# Basic Drawing





## Overview

- **Make a subclass of JPanel**
  ```
  public class MyPanel extends JPanel
  ```
- **Override the paintComponent method**
  ```
  @Override
  protected void paintComponent(Graphics g) { ... }
  ```
- **Call super.paintComponent**
  ```
  super.paintComponent(g);
  ```
  - Does default behavior like handling opacity
- **Use the supplied Graphics object to draw**
  ```
  g.drawLine(x1, y1, x2, y2);
  ```
  - (0,0) is *top* left corner, x goes to right, y goes *down*
- **Assign the panel as the content of the frame**
  ```
  frame.setContentPane(new MyPanel());
  ```

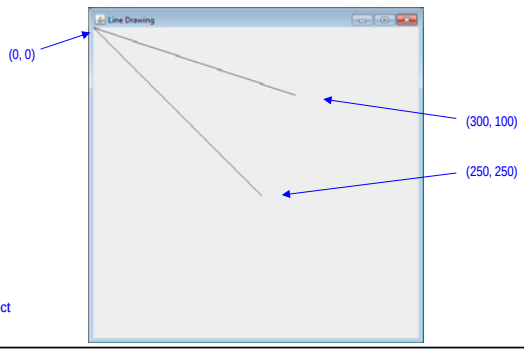
# Example: JFrame

```
public class LineFrame extends JFrame {
  public LineFrame() {
    super("Line Drawing");
    setContentPane(new LinePanel());
    setSize(500,500);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setVisible(true);
  }

  public static void main(String[] args) {
    new LineFrame();
  }
}
```



# Example: JPanel

```
public class LinePanel extends JPanel {
  @Override
  protected void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.drawLine(0, 0, 250, 250);
    g.drawLine(0, 0, 300, 100);
  }
}
```




- (0, 0) is top left corner.
- x goes to the right, as most people expect
- y goes down, contrary to what many people expect





# Basic Drawing Methods





## Overview

- **Drawing lines and shapes**
  - Use supplied Graphics object
    - g.drawLine, not drawLine
  - Understand coordinate system
    - (0,0) is top left corner, x goes to right, y goes down
  - Make outlines or solid shapes
    - g.drawRect: outline, g:fillRect: solid
- **Setting default features of JPanel**
  - Do not use Graphics object
    - setBackground, setForeground, setFont, etc.
    - Usually called in constructor, not in paintComponent
  - Remember to call super.paintComponent
    - Background colors have no effect unless window is opaque (default) and you call super.paintComponent

## Methods of Graphics Class

- **drawString(string, left, bottom)**
  - Draws a string in the current font and color with the *bottom left* corner of the string at the specified location
  - One of the few methods where the y coordinate refers to the bottom of shape, not the top. But y values are still with respect to the *top left* corner of the applet window
- **drawRect(left, top, width, height)**
  - Draws the outline of a rectangle (1-pixel border) in the current color
- **fillRect(left, top, width, height)**
  - Draws a solid rectangle in the current color
- **drawLine(x1, y1, x2, y2)**
  - Draws a 1-pixel-thick line from (x1, y1) to (x2, y2)



## Methods of Graphics Class

- **drawOval, fillOval**
  - Draws an outlined or solid oval, where the arguments describe a rectangle that bounds the oval
- **drawPolygon, fillPolygon**
  - Draws an outlined or solid polygon whose points are defined by arrays or a Polygon (a class that stores a series of points). By default, polygon is closed; to make an open polygon use the drawPolyline method
- **drawImage**
  - Draws an image
    - Image usually created with Toolkit.getDefaultToolkit().getImage – see upcoming example
  - Supports JPEG, GIF (including animated GIF), or PNG

## Methods of Graphics Class

- **setColor, getColor**
  - Specifies the foreground color prior to drawing operation
  - By default, the graphics object receives the foreground color of the window
    - As set via setForeground from the constructor
  - Java has 16 predefined colors (Color.RED, Color.BLUE, etc.) or create your own color: new Color(r, g, b)
  - Changing the color of the Graphics object affects only the drawing that explicitly uses that Graphics object
    - To make permanent changes, call the *panel's* setForeground method



## Methods of JPanel Class

- **getWidth, getHeight**
  - Returns current size of panel. Can change at run time.
- **setBackground**
  - Sets background color.
- **setForeground**
  - Sets default drawing color. You can also omit this and just set the color before drawing
    - g.setColor(Color.BLUE);
    - g.fillRect(…);
- **repaint**
  - Instructs the event handling thread to call paint. Used when you have a thread or event handler that changes some data structure that is used by paint.

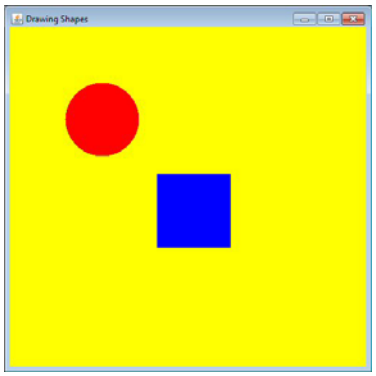
## Example: JFrame

```
public class ShapeFrame extends JFrame {
  public ShapeFrame() {
    super("Drawing Shapes");
    setContentPane(new ShapePanel(Color.YELLOW));
    setSize(500,500);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setVisible(true);
  }

  public static void main(String[] args) {
    new ShapeFrame();
  }
}
```

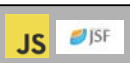


## Example: JPanel

```
public class ShapePanel extends JPanel {
  public ShapePanel(Color bgColor) {
    setBackground(bgColor);
  }

  @Override
  protected void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.setColor(Color.RED);
    g.fillOval(75, 75, 100, 100);
    g.setColor(Color.BLUE);
    g.fillRect(200, 200, 100, 100);
  }
}
```








# Overriding Lifecycle Methods





## Overview

- **Questions**
  - Why did we write paintComponent?
    - Since we never called it
  - Why did we use @Override?
    - Since omitting it in our example has no effect
- **Answers**
  - Because some *other* code calls it
    - Same reason we used toString earlier, even though we never called toString ourselves
  - Two reasons
    - To catch errors at compile time (see upcoming slides)
    - To document usage to other developers:
      meaning and usage of paintComponent is specified by parent class, not by me

# Example: JFrame

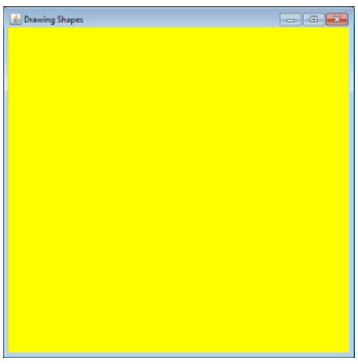
```
public class ShapeFrame extends JFrame {
  public ShapeFrame() {
    super("Drawing Shapes");
    setContentPane(new ShapePanel(Color.YELLOW));
    setSize(500,500);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setVisible(true);
  }

  public static void main(String[] args) {
    new ShapeFrame();
  }
}
```

This is the same JFrame code shown in last example.
But definition of the ShapePanel has changed slightly.



# Why Are No Shapes Drawn?

```
public class ShapePanel extends JPanel {
  public ShapePanel(Color bgColor) {
    setBackground(bgColor);
  }

  protected void paintcomponent(Graphics g) {
    super.paintComponent(g);
    g.setColor(Color.RED);
    g.fillOval(75, 75, 100, 100);
    g.setColor(Color.BLUE);
    g.fillRect(200, 200, 100, 100);
  }
}
```



# Fixing the Problem

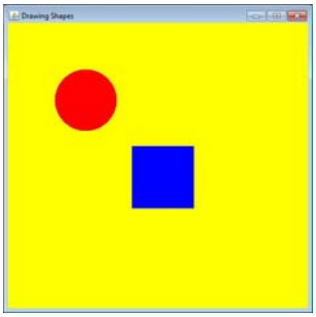
- **Fix 1: Use @Override**

```
@Override
protected void paintcomponent(Graphics g) { ... }
```

  - Code will fail at compile time, not work oddly at run time
    - "The method paintcomponent(Graphics) of type ShapePanel must override or implement a supertype method"
- **Fix 2: call it paintComponent**

```
@Override
protected void paintComponent(Graphics g) {
  ...
}
```

  - @Override has no effect other than documentation on *correct* code, but with *incorrect* code it helps catch errors at compile time







# Loading and Drawing Images

# Overview

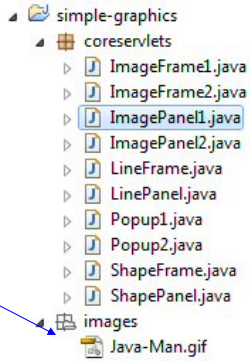
- **Basic syntax**
  - Define instance variable of type Image
    ```
    private Image image;
    ```
  - Use Toolkit to get the image (usually in constructor)
    ```
    image = Toolkit.getDefaultToolkit().getImage(…)
    ```
    - You can supply either a path relative to root of the code, or a URL
    - Resource can be JPEG, GIF, or PNG
  - Draw the image in paintComponent
    ```
    g.drawImage(image, left, top, windowImageDrawnOn);
    ```
- **Special notes**
  - getImage returns immediately; the image is downloaded in a background thread
  - If image is incomplete when paintComponent is called, paintComponent is automatically called again later
  - If you use a URL for the image location, you need a try/catch block



# Relative Path: JFrame

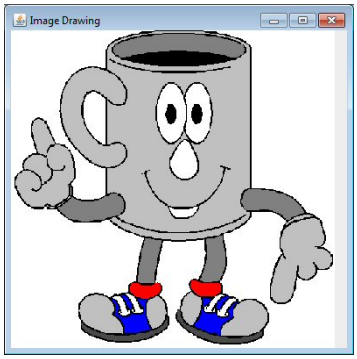
```
public class ImageFrame1 extends JFrame {
  public ImageFrame1(String relativePath) {
    super("Image Drawing");
    setContentPane(new ImagePanel1(relativePath));
    setSize(400,400);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setVisible(true);
  }

  public static void main(String[] args) {
    new ImageFrame1("images/Java-Man.gif");
  }
}
```

This address is relative to the root of the local classpath, not relative to the package-specific folder. So, in this case, the images folder is in the root of the Eclipse project. If you have separate folders for source files and class files, then the images folder would go in the root of the bin (not src) folder.



## Relative Path: JPanel

```
public class ImagePanel1 extends JPanel {
  private Image image;

  public ImagePanel1(String relativePath) {
    image = Toolkit.getDefaultToolkit().getImage(relativePath);
  }

  @Override
  protected void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.drawImage(image, 0, 0, this);
  }
}
```






## Absolute URL: JFrame

```
public class ImageFrame2 extends JFrame {
  public ImageFrame2(String urlString) {
    super("Image Drawing");
    setContentPane(new ImagePanel2(urlString));
    setSize(560,420);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setVisible(true);
  }

  public static void main(String[] args) {
    new ImageFrame2("http://www.coreservlets.com/images/" +
                    "marty-hall-java-license-plate.jpg");
  }
}
```

# Absolute URL: JPanel

```
public class ImagePanel2 extends JPanel {
  private Image image;

  public ImagePanel2(String urlString) {
    try {
      URL imageUrl = new URL(urlString);
      image = Toolkit.getDefaultToolkit().getImage(imageUrl);
    } catch(MalformedURLException mfe) {
      System.err.println("Illegal URL: " + mfe);
    }
  }

  @Override
  protected void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.drawImage(image, 0, 0, this);
  }
}
```






# Basic Try/Catch Blocks

## (More Features in Section on File IO)

# Exceptions and try/catch Blocks

- **Handling exceptions**
  - If your code potentially has an error (i.e., "throws an exception"), you must either use a try/catch block or throw the exception, and let the calling code handle it.
    - There are a few error types (e.g., NullPointerException, ArrayIndexOutOfBoundsException) which you are not required to handle. These error types are all subtypes of RuntimeException.
- **Basic form**

```
try {
  statement1;
  statement2;
  ...
} catch(SomeExceptionClass someVar) {
  handleTheException(someVar);
}
```



# More Info

| One Catch | Multiple Catches | Throwing |
|---|---|---|
| try {<br>  statement1;<br>  statement2;<br>  ...<br>} catch(Eclass var) {<br>  doBlah(var);<br>}<br><br>Run all statements in try block. If everything finishes normally, you are done. If an exception of type Eclass occurs, jump down to catch block. | try {<br>  statement1;<br>  statement2;<br>  ...<br>} catch(Eclass1 var1) {<br>  ...<br>} catch(Eclass2 var2) {<br>  ...<br>} catch(Eclass3 var3) {<br>  ...<br>}<br>First matching catch fires, so exceptions should be ordered from most specific to most general<br>• Due to inheritance, more than one match is possible | public void foo()<br>  throws Eclass1 {<br>  …<br>}<br><br>Instead of catching exception here, throw it to the calling code, which must catch the exception or throw it further. |

## Preview of Topics Covered in File I/O Lecture

**Covered here: basics**

```
try {
  statement1;
  statement2;
  ...
} catch(Eclass1 var1) {
  ...
} catch(Eclass2 var2) {
  ...
} catch(Eclass3 var3) {
  ...
}
...
```

**Covered later: finally**

```
try {...
} catch(...) {...
} finally {
  ...
}
```

**Covered later: multicatch**

```
try {...
} catch(Eclass1 | Eclass e) {
  ...
} ...
```

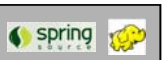
**Covered later: try with resources**

```
try (SomeAutoCloseable var = ...) {...
} catch(...) { ...
} ...
```








# Other Graphics Capabilities



## Overview

- **We just scratched the surface**
  - *Very* simple form of window
  - *Very* simple type of graphics
- **Motivation**
  - Course focuses on topics that are applicable to all Java apps (server-side apps, desktop apps, phone apps)
    - This type of graphics applies only to desktop apps
  - But, learning very simple graphics has some benefits for all Java developers
    - Makes exercises more interesting
    - Gives idea of what desktop and phone developers do
    - Gives excuse to cover lifecycle methods, overriding, methods that run in the background, try/catch blocks, and inner classes (next lecture)



## Other Graphics Capabilities in Java SE

- **Lots of GUI controls**
  - Buttons, sliders, checkboxes, menus, tables, etc.
- **Subwindows and layout managers**
  - For organizing components in the page
- **Rich 2D drawing**
  - Line properties, translucent drawing, rotating and scaling images, coordinate transformations, etc.
    - 3D graphics requires separate library, not part of Java SE
- **Animation**
  - Double-buffered drawing combined with threads
- **All these topics are covered on Web site**
  - In section “Desktop Graphics in Java”

# Wrap-Up





## Summary: Graphics-Specific Topics

- **Pop up JFrame**
  - Extend JFrame
    - Give frame a title, size, close operation, setVisible(true)
- **Assign a JPanel for the content**

```
setContentPane(new MyPanel(...));
```

- **Draw in the paintComponent method of the JPanel**

```
@Override
protected void paintComponent(Graphics g) {
  super.paintComponent(g);
  g.drawLine(x1, y1, x2, y2);
  g.drawImage(image, left, top, this);
}
```

# Summary: General Topics

- **Use try/catch blocks to handle exceptions**

```
try {
  ...
} catch(Eclass1 var1) { ...
} catch(Eclass2 var2) { ...
} catch(Eclass3 var3) { ...
}
```

  - More features of try/catch blocks in later lecture (File I/O)

- **Use @Override for replacing parent methods**
  - Catches typos at compile time
    - If does not match, code will not compile
  - Expresses design intent
    - Tells fellow developers that meaning and usage of method comes from parent class