GUI in Java (Advanced)

Topics:

including

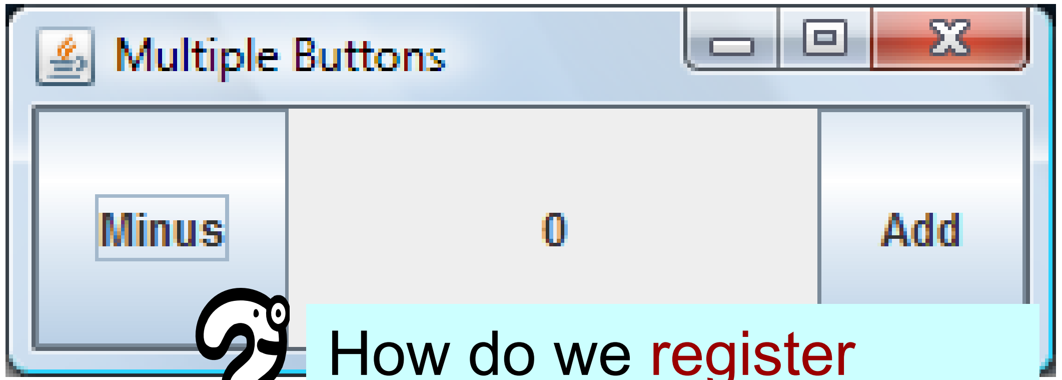
- Event Handling (again) // Using Anonymous Inner Classes
- Graphics in GUI (**`paintComponent()`**, **`Graphics2D`**)
- Other related classes: **`Color`**, **`Font`**, **`FontMetrics`**
- Animation using Inner Classes

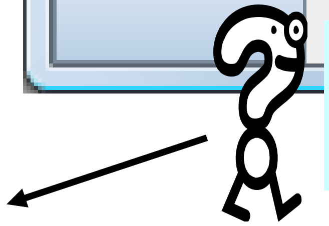
Chapters 8, 17 – "Big Java" book
Chapters 12, 13 – "Head First Java" book
Chapters 12–14 – "Introduction to Java Programming" book
Chapter 4 – "Java in a Nutshell" book

Queen Mary
University of London

Handling Multiple Events (1/2)

- How do we deal with events from multiple widgets?
  - *Example GUI*: Clicking the Add button should add 1 to the number in the middle; clicking the Minus button should subtract 1 from the number.





How do we register listeners on two buttons?

- Ways to deal with *multiple* event sources:

1. Register each widget with the required listener and then determine which widget generated the event.
2. Use an *anonymous inner class* for each event source.
3. Use a specialised *inner class* for each event source.

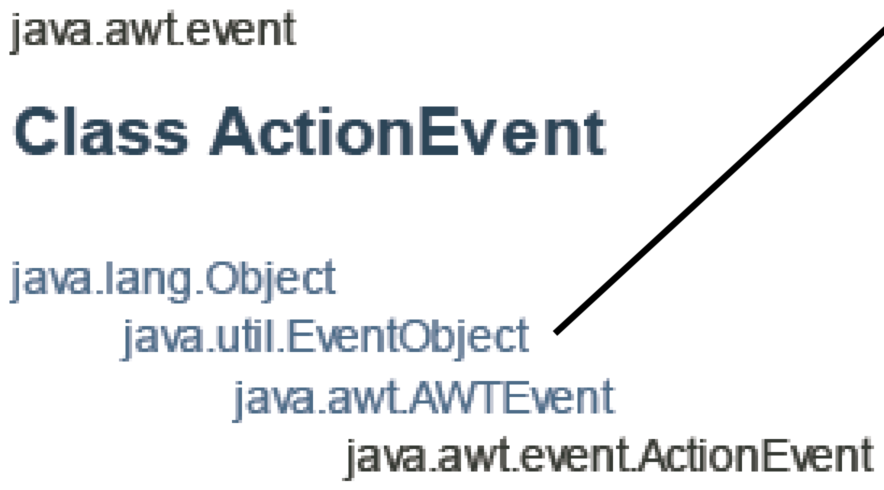
1 Setting up the GUI

```
public class MultipleButtons extends JFrame implements
                                             ActionListener {
  private JButton addButton,minusButton;
  private JLabel label;
  private int number;
  public MultipleButtons() {
    super("Multiple Buttons");
    addButton = new JButton("Add");
    addButton.addActionListener(this);
    minusButton = new JButton("Minus");
    minusButton.addActionListener(this);
    label = new JLabel(""+ number, JLabel.CENTER);
    this.getContentPane().add(this.addButton, BorderLayout.EAST);
    this.getContentPane().add(this.label, BorderLayout.CENTER);
    this.getContentPane().add(this.minusButton, BorderLayout.WEST);
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    this.setSize(400, 100);
    this.setVisible(true);
  }
```

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

1 Determining the Event Source

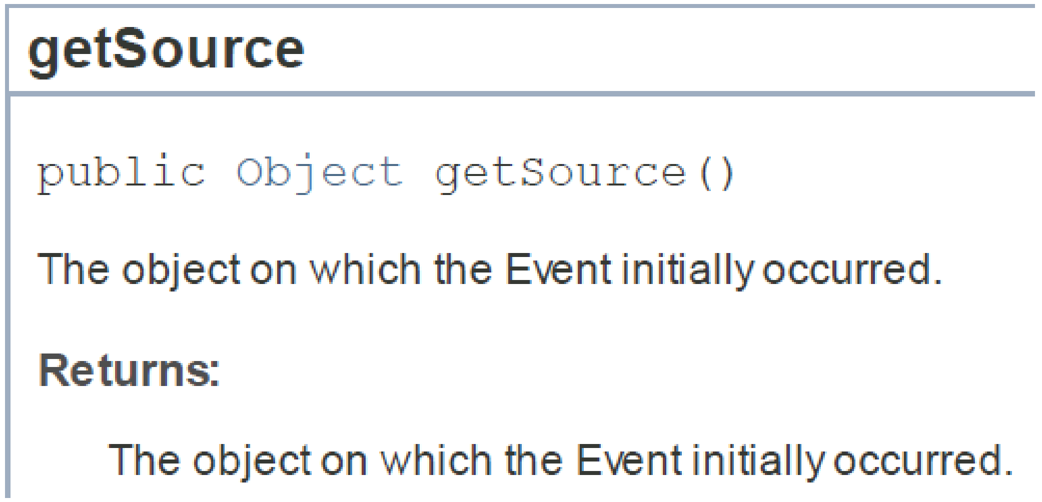
```
  public void actionPerformed(ActionEvent e){
    // How do we figure out which JButton is which?
    JButton eventSource = (JButton) e.getSource();
    if (eventSource.equals(addButton))
      label.setText("" + (++number));
    else if (eventSource.equals(minusButton))
      label.setText("" + (--number));
  }
}
```



java.awt.event

Class ActionEvent

```
java.lang.Object
    java.util.EventObject
        java.awt.AWTEvent
            java.awt.event.ActionEvent
```



getSource

`public Object getSource()`

The object on which the Event initially occurred.

Returns:

The object on which the Event initially occurred.

http://docs.oracle.com/javase/8/docs/api/java/awt/event/ActionEvent.html

2 Using Anonymous Inner Classes (1/3)

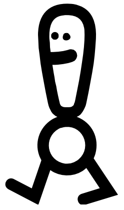
- An anonymous class is a special kind of class: a *local class without a name*.
  - It allows an object to be created using an expression that *combines* object creation with the declaration of the class.
  - This *avoids naming a class* but:
    - only one instance of the class can ever be made;
    - class can't be accessed from anywhere else in the program.
- An anonymous class is defined as *part of a new expression* and *must* be a subclass or implement an interface.
  - The class body can define methods but cannot define any constructors.

Using Anonymous Inner Classes (2/3)

```
public class MultipleButtons extends JFrame {
  private JButton addButton,minusButton;
  private JLabel label;
  private int number;

  public MultipleButtons() {
    super("Multiple Buttons");
    addButton = new JButton("Add");
    addButton.addActionListener(
      new ActionListener() {
        public void actionPerformed(ActionEvent e) {
          label.setText("" + (++number));
        }
      }
    );
```

No `implements` keyword.



We *only deal* with what we would like to do when the `addButton` is pressed; the `minusButton` will have its own class.

Class declaration is actually included in between the open and close brackets.

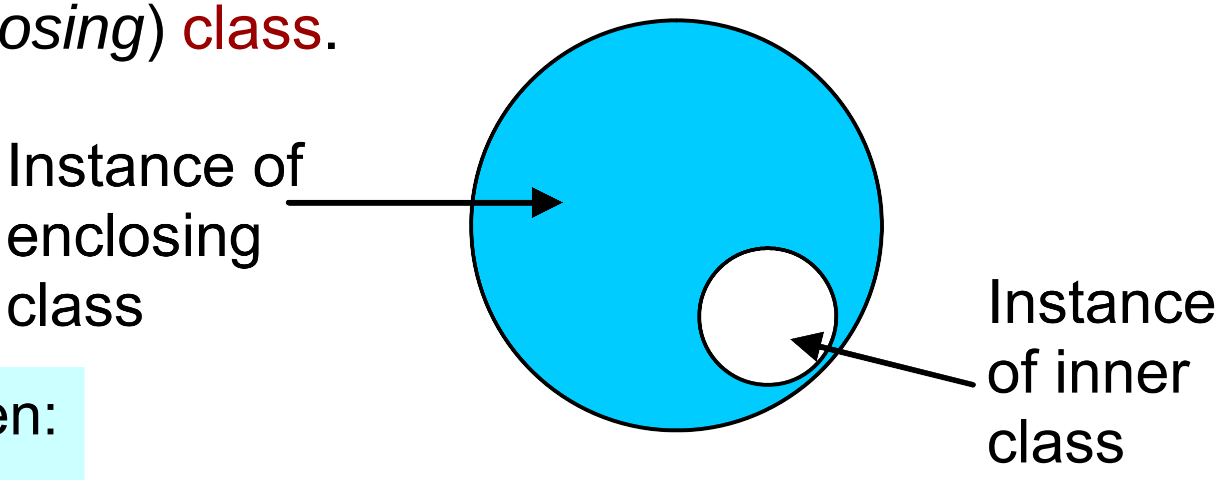
2 Using Anonymous Inner Classes (3/3)

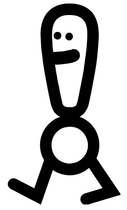
```
    minusButton = new JButton("Minus");
    minusButton.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        label.setText("" + (--number));
      }
    });
    label = new JLabel(""+ number, JLabel.CENTER);
    this.getContentPane().add(this.addButton, BorderLayout.EAST);
    this.getContentPane().add(this.label, BorderLayout.CENTER);
    this.getContentPane().add(this.minusButton, BorderLayout.WEST);
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    this.setSize(400, 100);
    this.setVisible(true);
  }
  public static void main(String[] args) { new MultipleButtons(); }
}
```

Class for the `Minus` button.

3 Inner Classes

- Inner classes are *named versions of anonymous classes*.
  - Yes, you were told that you could only have one class per file but that isn't strictly true! ☺
  - In some cases (and this is one of them), you can have more than one class in the same file.
- Inner (or *Nested*) Class: Standard class declared within the scope of a standard top-level (or *enclosing*) class.



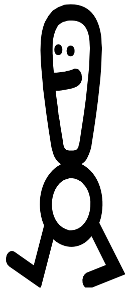


Until now, the rule has been: one class per **`.java`** file.

3 Example: Inner Class

```
public class OuterClass {
  private int data;

  /** A method in the outer class. */
  public void methodOuter() {
    // Do something.
    InnerClass myInnerClass = new InnerClass();
  }
  public class InnerClass {
    /** A method in the inner class. */
    public void methodInner() {
      // Directly reference data & method
      // defined in its outer class.
      data++;
      methodOuter();
    }
  }
}
```



An *inner class* is defined in the scope of an outer class. (An instance of) it can reference data and methods (even private ones) of the *outer class* it belongs to.

3 Instance of an Inner Class

- An instance of an inner class (i.e. an *inner object*) must be associated with a specific outer object on the heap!
  - You instanciate an *inner* class from within an *outer* class: this means that the inner object will have a special “link” (or bond) with a specific instance of the outer class.
  - Instantiation of the inner class is done in the usual way …
  - Example:

```
public class OuterClass {
  private int data;
  MyInnerClass myInner = new MyInnerClass();
  public void methodOuter() { myInner.methodInner(); }

  public class MyInnerClass {
    public void methodInner() { data = 10; }
  }
}
```

`methodInner()` can use outer class private variable as if that variable belonged to `MyInnerClass`.

3 Using Inner Classes (1/2)

```
public class MultipleButtons extends JFrame {
  private JButton addButton,minusButton;
  private JLabel label;
  private int number;
  public MultipleButtons() {
    super("Multiple Buttons");
    addButton = new JButton("Add");
    addButton.addActionListener(new PlusListener());
    minusButton = new JButton("Minus");
    minusButton.addActionListener(new MinusListener());
    label = new JLabel(""+ number, JLabel.CENTER);
    this.getContentPane().add(this.addButton, BorderLayout.EAST);
    this.getContentPane().add(this.label, BorderLayout.CENTER);
    this.getContentPane().add(this.minusButton, BorderLayout.WEST);
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    this.setSize(400, 100);
    this.setVisible(true);
  }
```

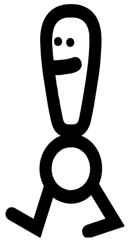
No `implements` keyword.

Instances of *inner classes* used for handling events.

3 Using Inner Classes (2/2)

```
  public class PlusListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      label.setText(""+(++number));
    }
  }
  public class MinusListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      label.setText(""+(--number));
    }
  }
  public static void main(String[] args) { new MultipleButtons(); }
}
```

Two inner classes: one for *adding* and one for *subtracting*.



No need to override the method `actionPerformed()`; this is now dealt with in the inner classes above.

… and things for you to try out!

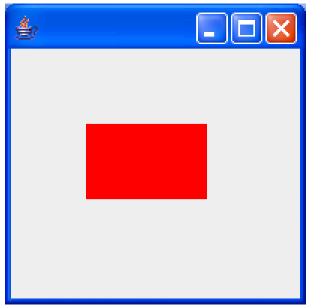
Three types of graphics in GUI

- **Putting things on a GUI**:
  - Add widgets to a frame (*review only!*): e.g. add buttons, menus

```
JFrame myFrame = new JFrame();
myFrame.getContentPane().add(myButton);
```

  - Draw 2D graphics on a widget: paint shapes with a graphics' object

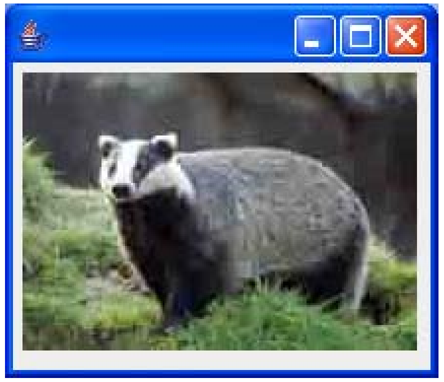
```
Graphics myGraphics;
myGraphics.fillRect(50,50,80,50);
```



  - Put a JPEG on a widget: add pictures

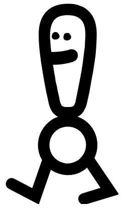
```
Graphics myGraphics;
Image myImage = new ImageIcon("badger.jpg").getImage();
myGraphics.drawImage(myImage,10,10,this);
```

(x,y) coordinates relative to the widget, not the frame



Paintable Widgets *or* How to Draw on GUIs

- To put graphics on the screen:
  - **Step 1**: Make a paintable widget.
    - Create subclass of **`JPanel`** & override the **`paintComponent()`** method.
    - Put all the graphics code in the **`paintComponent()`** method.
    - The **`paintComponent()`** method is called only by the JVM; the *programmer does not call it*!
    - It takes a **`Graphics`** object – drawing canvas for what is displayed on the screen.
  - **Step 2**: Add widget to frame.



We already know how to do Step 2!

```
import java.awt.*;
import javax.swing.*;
class MyDrawingPanel extends JPanel {
  public void paintComponent(Graphics g) {
    g.setColor(Color.red);
    g.fillRect(50,50,80,50);
  }
}
```

Example: Placing Graphics on a GUI

AddGraphics.java

```
import javax.swing.*;

public class AddGraphics {
  public static void main(String[] args) {
    JFrame myFrame = new JFrame();
    MyDrawingPanel myDrawingPanel = new MyDrawingPanel();
    myFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    myFrame.getContentPane().add(myDrawingPanel);
    myFrame.setSize(220, 190);
    myFrame.setVisible(true);
  }
}
```

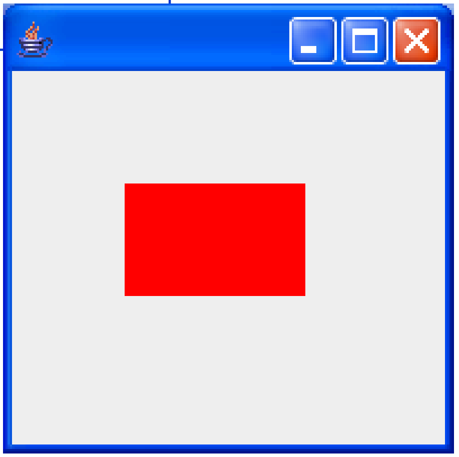
MyDrawingPanel.java

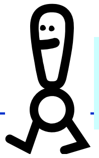
```
import java.awt.*;
import javax.swing.*;

public class MyDrawingPanel extends JPanel {
  public void paintComponent(Graphics g) {
    g.setColor(Color.red);
    g.fillRect(50,50,80,50);
  }
}
```

Output is …

```
> java AddGraphics
```

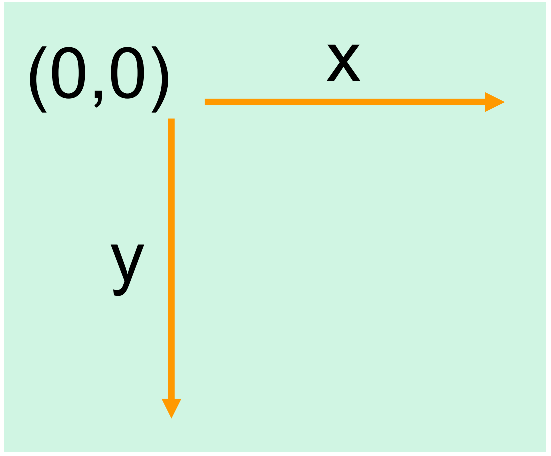




Try placing a picture on the GUI!

(More about) `Graphics` Methods (1/2)

- `Graphics` class: Java graphics are based on *pixels* (small dot on the screen that can be accessed).
  - Different screens have different pixel counts.
  - A *pixel* is identified by a pair of numbers (*coordinates*) starting at zero, (x,y):
    - x = horizontal position (increases left to right)
    - y = vertical position (increases top to bottom)



- Examples:

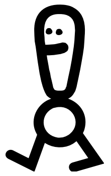
(1) `drawString(string,x,y)` → draw string starting at position `(x,y)`.

(2) `drawRect(x,y,width,height)` → draw rectangle at `(x,y)` with given `width` and `height`.

(3) `fillRect(x,y,width,height)` → same as (2), but fill.

(More about) Graphics Methods (2/2)

- Other examples:



See the Java API for more **`Graphics`** methods!

`(4) drawOval(x,y,width,height)` → draw oval with **`(x,y,width,height)`**.

`(5) fillOval(x,y,width,height)` → same as (4), but fill solid.

`(6) drawLine(x1,y1,x2,y2)` → draw line from **`(x1,y1)`** to **`(x2,y2)`**.

`(7) drawArc(x,y,width,height,startAngle,sweepAngle)` → same as (4) but start at **`startAngle`**, sweep degrees defined by **`sweepAngle`**.

Example:

```
drawArc(30,30,230,230,0,180) // draw 1/2 cycle on top
```

`(8) fillArc(x,y,width,height,startAngle,sweepAngle)` → same as (7) except fill the sweeping region.

`(9) drawPolygon(int[] X,int[] Y,int z)` → **X,Y** defines **`z`** points of the polygon *w.r.t.* **`(x,y)`** coordinates.

`(10)fillPolygon(int[] X,int[] Y,int z)` → same as (9), but fill.

Example: Drawing Polygons

- For polygons, we also can do the following:

```
Polygon myPentagon = new Polygon();
myPentagon.addPoint(a1,b1);
myPentagon.addPoint(a2,b2);
myPentagon.addPoint(a3,b3);
myPentagon.addPoint(a4,b4);
myPentagon.addPoint(a5,b5);
g.drawPolygon(myPentagon);
```

  – Here, **`(a1,b1)`**, **`(a2,b2)`**, **`(a3,b3)`**, **`(a4,b4)`**, **`(a5,b5)`** are *polygon vertices*. This is equivalent to:

```
int[] X = {a1,a2,a3,a4,a5};
int[] Y = {b1,b2,b3,b4,b5};
g.drawPolygon(X,Y,5);
```

… and things for you to try out!

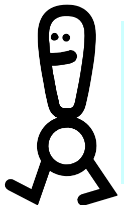
The Graphics2D Class

- **`public void paintComponent(Graphics g) {...}`**
  - **`g`** is a **`Graphics`** object → using polymorphism, **`g`** can be an instance of a subclass of **`Graphics`**;
  - **`g`** is actually an instance of the **`Graphics2D`** class.
  - If you need to use a method from **`Graphics2D`** class, you can't use the **`paintComponent()`** method's **`g`** parameter directly; instead,

```
Graphics2D g2d = (Graphics2D) g;
```

- (Some of the) methods in **`Graphics2D`** class:

```
fill3DRect()    rotate()
draw3DRect()    scale()
```



See the Java API for more **`Graphics2D`** methods!

Tutorial on 2D graphics:
http://docs.oracle.com/javase/tutorial/2d/index.html

Color Class

static variable	RBG value		
Color.black	R:0,	G:0,	B:0
Color.blue	R:0,	G:0,	B:255
Color.cyan	R:0,	G:255,	B:255
Color.gray	R:128,	G:128,	B:128
Color.green	R:0,	G:255,	B:0
Color.magenta	R:255,	G:0,	B:255
Color.red	R:255,	G:0,	B:0
Color.white	R:255,	G:255,	B:255
Color.yellow	R:255,	G:255,	B:0

- Java has a **`Color`** class.
  - To define the colour of an object, you can directly use the static colour variables of the **`Color`** class.
- Example:

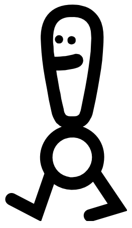
```
public void paintComponent(Graphics g) {
   g.setColor(Color.red);     // g object becomes red
   g.drawLine(10,10,200,200); // draw a red line
}
```

- You can also set your own colour by choosing an RGB value:

```
Color myColor = new Color(r,g,b);
```

  where each of the values **r**, **g** and **b** varies from *0* to *255*.
- The **`Color`** class has its own methods like: **`getRed()`**, **`getGreen()`**, **`getBlue()`**.

```
int c = myColor.getRed(); // c = 255
```

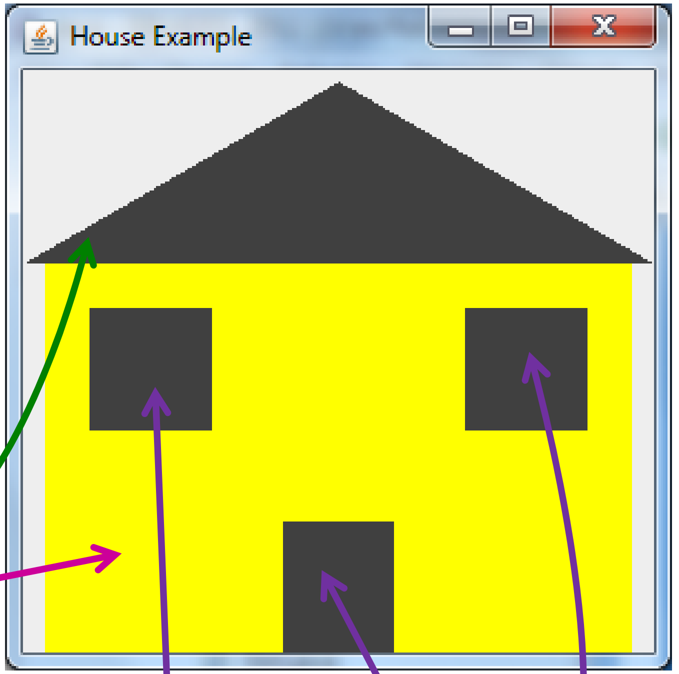


See the **`Color`** class on the Java API!

Example: Drawing on a GUI (1/2)

```
import javax.swing.*; import java.awt.*;
public class HousePanel extends JPanel {
  public void paintComponent(Graphics g) {
    int houseX = 10;
    int houseY = getHeight()/3;
    int door = 50, window = 55, windowInset = 20;
    int houseWidth = getWidth() - (houseX*2);
    int houseHeight = getHeight() - 50;
    int[] x = {0, getWidth()/2, getWidth()};
    int[] y = {getHeight()/3, 5, getHeight()/3};
    g.setColor(Color.darkGray);
    g.fillPolygon(x, y, 3);
    g.setColor(Color.yellow);
    g.fillRect(houseX, houseY, houseWidth, houseHeight);
    g.setColor(Color.darkGray);
    g.fillRect(houseX+windowInset, houseY+windowInset, window, window);
    g.fillRect(houseX+houseWidth-windowInset-window,
           houseY+windowInset, window, window);
    g.fillRect(houseX+(houseWidth/2)-door/2, (houseHeight/2)+houseY+windowInset/2,
           door, houseHeight/2-windowInset/2);
  }
  public static void main(String[] args) {
    // code for main()
  }
}
```

```
JFrame frame = new JFrame("House Example");
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.getContentPane().add(new HousePanel());
frame.setSize(300,300);
frame.setVisible(true);
```



Queen Mary University of London

… and things for you to try out!

Font and FontMetrics Classes (1/2)

- **`java.awt.Font`**: Specifies fonts for text and drawings in GUIs.
  - Create **`Font`** object before setting the font:

    ```
    Font f = new Font("TimesRoman", Font.BOLD, 18);
    g.setFont(f);
    ```

  - Arguments to the **`Font`** constructor:
    - Font *name*:
      - Logical *font name*: e.g. **`Monospaced`**, **`Serif`**, **`SansSerif`**, or **`Symbol`**. ***or***
      - Font *face name*, e.g. **`"Helvetica Bold"`**.
    - Font *style*: e.g. **`Font.BOLD`**, **`Font.ITALIC`**, **`Font.PLAIN`**
    - Point *size*.

Font and FontMetrics Classes (2/2)

- **`java.awt.FontMetrics`**: Abstract class to get properties of fonts.
  - **Example**:

```
Graphics g;
// other code ...
Font f = new Font("Serif", Font.PLAIN, 12);
g.setFont(f);
FontMetrics fm = g.getFontMetrics();
int a = fm.getAscent();
int b = fm.getMaxAdvance();
```

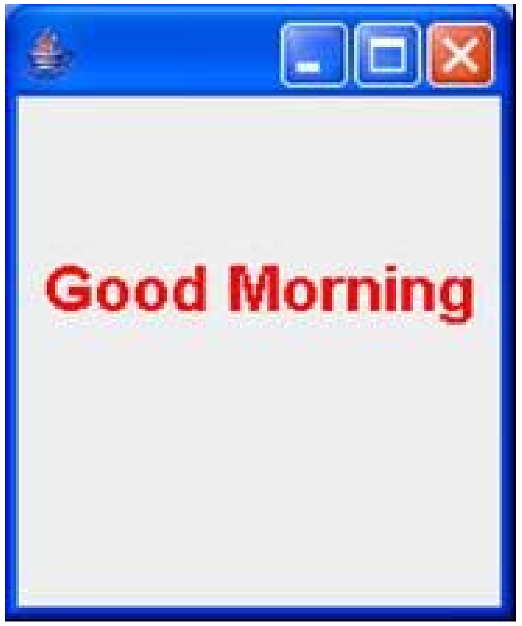
  - You can use font metrics for text placement e.g.,

```
g.drawString("Hello World",
  this.getWidth()/2 – fm.stringWidth("Hello World")/2,
  this.getHeight()/2 - fm.getHeight()/2);
```

Example: Testing Font Classes

```
import java.awt.*;
import javax.swing.*;

public class TestingFontClasses extends JPanel {
  public void paint(Graphics g) {
    int fontSize = 20;
    String s = "Good Morning";
    Font f = new Font(s, Font.BOLD, fontSize);
    g.setFont(f);
    FontMetrics fm = g.getFontMetrics();
    g.setColor(Color.red);
    g.drawString(s, this.getWidth()/2 – fm.stringWidth(s)/2,
                 this.getHeight()/2 - fm.getHeight()/2);
  }
  public static void main(String[] args) {
    JFrame frame = new JFrame();
    frame.getContentPane().add(new TestingFontClasses());
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setSize(160,200);
    frame.setVisible(true);
  }
}
```



Output …

Example: Graphics and Event Handling in a GUI (1/2)

MyDrawingPanel.java

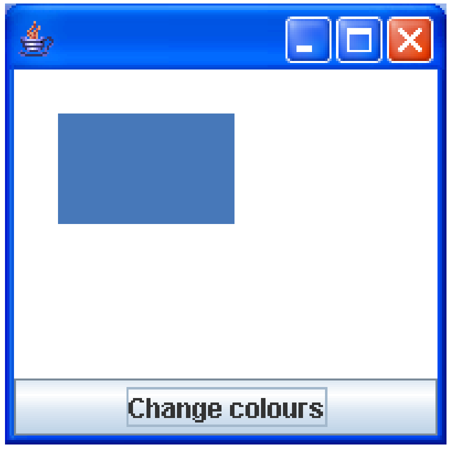
```
import java.awt.*;
import javax.swing.*;

public class MyDrawingPanel extends JPanel {
  public void paintComponent(Graphics g) {
    g.setColor(Color.white);
    g.fillRect(0,0,this.getWidth(),this.getHeight());
    int red = (int) (Math.random()*255);
    int green = (int) (Math.random()*255);
    int blue = (int) (Math.random()*255);
    Color randomColor = new Color(red,green,blue);
    g.setColor(randomColor);
    g.fillRect(20,20,80,50);
  }
}
```

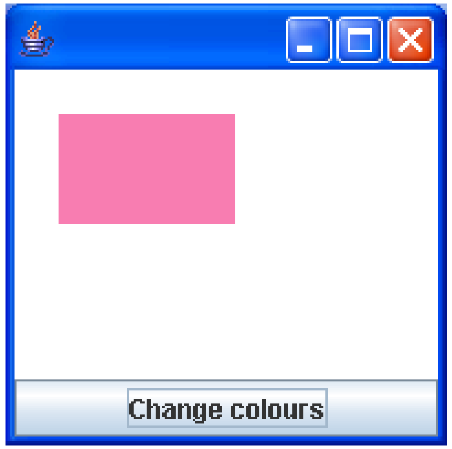
(*) user clicks button

Expected behaviour …

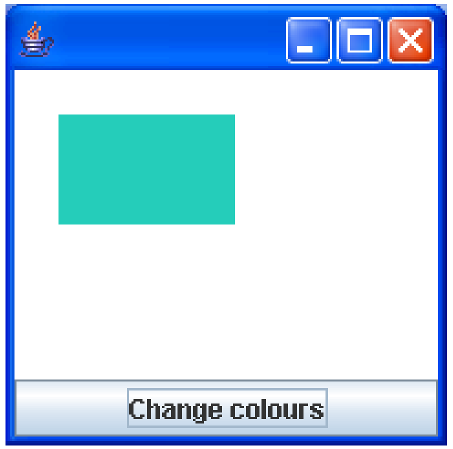
```
> java SimpleGuiV3
```



(*)



(*)



Example: Graphics and Event Handling in a GUI (2/2)

SimpleGuiV3.java

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class SimpleGuiV3 implements ActionListener {
  JFrame myFrame;
  public static void main(String[] args) {
    SimpleGuiV3 myGui = new SimpleGuiV3();
    myGui.go();
  }
  public void go() {
    myFrame = new JFrame();
    myFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    JButton myButton = new JButton("Change colours");
    myButton.addActionListener(this);
    MyDrawingPanel myDrawingPanel = new MyDrawingPanel();
    myFrame.getContentPane().add(BorderLayout.SOUTH, myButton);
    myFrame.getContentPane().add(BorderLayout.CENTER, myDrawingPanel);
    myFrame.setSize(200, 200);
    myFrame.setVisible(true);
  }
  public void actionPerformed(ActionEvent event) {
    myFrame.repaint();
  }
}
```

calls **`paintComponent()`** method when user clicks the button

… and things for you to try out!

Example: Animation using Inner Classes (1/2)

- Problem: Paint a square where the square moves across the screen from top left hand corner to bottom right hand corner.
- How simple animation can be implemented:
  - Step 1: Paint object at coordinate (x,y).
  - Step 2: Repaint object at a different coordinate (x,y).
  - Step 3: Repeat *Step 2* for however long the animation is to last.
  - This can be done by using inner classes.

It would be nice to have **`(x,y)`** values change every time **`paintComponent()`** gets called.

```
import java.awt.*;
import javax.swing.*;

public class MyDrawingPanel extends JPanel {
  public void paintComponent(Graphics g) {
    g.setColor(Color.red);
    g.fillRect(x,y,50,50);
  }
}
```

Example: Animation using Inner Classes (2/2)

```
import javax.swing.*;
import java.awt.*;
public class SimpleAnimation {
  int x = 50;
  int y = 50;
  public static void main(String[] args) {
    SimpleAnimation myGui = new SimpleAnimation();
    myGui.go();
  }
  public void go() {
    JFrame myFrame = new JFrame();
    myFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    MyDrawingPanel myDrawingPanel = new MyDrawingPanel();
    myFrame.getContentPane().add(myDrawingPanel);
    myFrame.setSize(300,300);
    myFrame.setVisible(true);
    for (int i=0; i<130; i++) {
      x++;
      y++;
      myDrawingPanel.repaint();
(*)   try { Thread.sleep(50); }
      catch (Exception ex) { }
    }
  }
```

(*) To slow the repainting. Don't need to know about **Threads** for now …

inner class

```
  // (cont.)
  public class MyDrawingPanel extends JPanel {
    public void paintComponent(Graphics g) {
      g.setColor(Color.white);
      g.fillRect(0,0,this.getWidth(),this.getHeight());
      g.setColor(Color.red);
      g.fillRect(x,y,50,50);
    }
  }
}
```

… and things for you to try out!