

Java Programming Tutorial

Custom Graphics

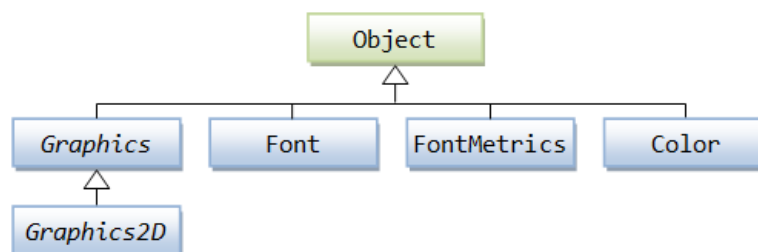
This chapter shows you how you can paint your own custom drawing (such as graphs, charts, drawings and, in particular, computer games) because you cannot find standard GUI components that meets your requirements. I shall stress that you should try to reuse the standard GUI components as far as possible and leave custom graphics as the last resort. Nonetheless, custom graphics is curial in game programming.

Read "Swing Tutorial" trail "[Performing Custom Painting](#)".

1. java.awt.Graphics Custom Painting

A *graphics context* provides the capabilities of drawing on the screen. The graphics context maintains states such as the color and font used in drawing, as well as interacting with the underlying operating system to perform the drawing. In Java, custom painting is done via the `java.awt.Graphics` class, which manages a graphics context, and provides a set of *device-independent* methods for drawing texts, figures and images on the screen on different platforms.

The `java.awt.Graphics` is an abstract class, as the actual act of drawing is system-dependent and device-dependent. Each operating platform will provide a subclass of `Graphics` to perform the actual drawing under the platform, but conform to the specification defined in `Graphics`.



1. Graphics Class's Drawing Methods

The `Graphics` class provides methods for drawing three types of graphical objects:

1. Text strings: via the `drawString()` method. Take note that `System.out.println()` prints to the system console, not to the graphics screen.
2. Vector-graphic primitives and shapes: via methods `drawXxx()` and `fillXxx()`, where `Xxx` could be `Line`, `Rect`, `Oval`, `Arc`, `PolyLine`, `RoundRect`, or `3DRect`.
3. Bitmap images: via the `drawImage()` method.

```

// Drawing (or printing) texts on the graphics screen:
drawString(String str, int xBaselineLeft, int yBaselineLeft);

// Drawing lines:
drawLine(int x1, int y1, int x2, int y2);
drawPolyline(int[] xPoints, int[] yPoints, int numPoint);

// Drawing primitive shapes:
drawRect(int xTopLeft, int yTopLeft, int width, int height);
drawOval(int xTopLeft, int yTopLeft, int width, int height);
drawArc(int xTopLeft, int yTopLeft, int width, int height, int startAngle, int arcAngle);
draw3DRect(int xTopLeft, int yTopLeft, int width, int height, boolean raised);
drawRoundRect(int xTopLeft, int yTopLeft, int width, int height, int arcWidth, int arcHeight);
drawPolygon(int[] xPoints, int[] yPoints, int numPoint);

// Filling primitive shapes:

```

TABLE OF CONTENTS

1. The `java.awt.Graphics` Class:
 - 1.1 Graphics Class' Drawing Methods
 - 1.2 Graphics Class' Methods for
 - 1.3 Graphics Class' Other Methods
 - 1.4 Graphics Coordinate System
2. Custom Painting Template
 - 2.1 Refreshing the Display via repaint
3. (Optional) Colors and Fonts
 - 3.1 `java.awt.Color`
 - 3.2 `java.awt.Font`
 - 3.3 `java.awt.FontMetrics`
4. Example 1: Moving an Object via
5. Example 2: Moving Sprites
6. Example 3: Paint
7. Drawing Images
 - 7.1 `javax.swing.ImageIcon`
 - 7.2 Graphics Class' `drawImage()`
8. Animation
 - 8.1 Animation using `javax.swing`
 - 8.2 Animation using a new Thread
9. (Advanced) A Closer Look at repaint



```

fillRect(int xTopLeft, int yTopLeft, int width, int height);
fillOval(int xTopLeft, int yTopLeft, int width, int height);
fillArc(int xTopLeft, int yTopLeft, int width, int height, int startAngle, int arcAngle);
fill3DRect(int xTopLeft, int yTopLeft, int width, int height, boolean raised);
fillRoundRect(int xTopLeft, int yTopLeft, int width, int height, int arcWidth, int arcHeight);
fillPolygon(int[] xPoints, int[] yPoints, int numPoint);

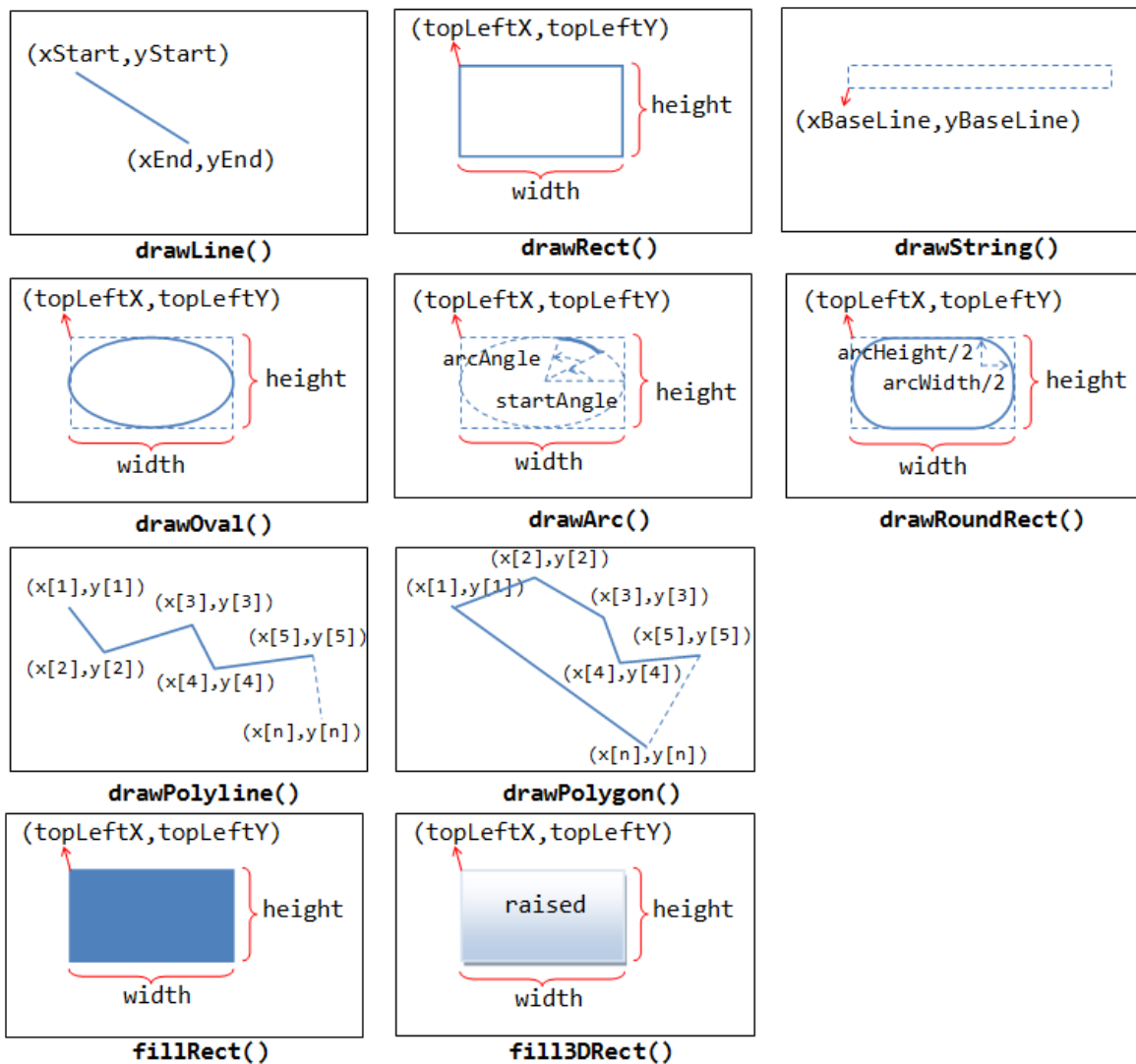
```

```

// Drawing (or Displaying) images:
drawImage(Image img, int xTopLeft, int yTopLeft, ImageObserver obs); // draw image with its size
drawImage(Image img, int xTopLeft, int yTopLeft, int width, int height, ImageObserver o); // resize image on screen

```

These drawing methods is illustrated below. The drawXxx() methods draw the outlines; while fillXxx() methods fill the internal. Shapes with negative *width* and *height* will not be painted. The drawImage() will be discussed later.



1. Graphics Class's Methods for Maintaining the Graph

The graphic context maintains *states* (or *attributes*) such as the current painting color, the current font for drawing text strings, and the current painting rectangular area (called *clip*). You can use the methods `getColor()`, `setColor()`, `getFont()`, `setFont()`, `getClipBounds()`, `setClip()` to get or set the color, font, and clip area. Any painting outside the clip area is ignored.

```

// Graphics context's current color.
void setColor(Color c)
Color getColor()

// Graphics context's current font.
void setFont(Font f)
Font getFont()

// Set/Get the current clip area. Clip area shall be rectangular and no rendering is performed outside the clip area.
void setClip(int xTopLeft, int yTopLeft, int width, int height)
void setClip(Shape rect)
public abstract void clipRect(int x, int y, int width, int height) // intersects the current clip with the given rectangle
Rectangle getClipBounds() // returns an Rectangle
Shape getClip() // returns an object (typically Rectangle) implements Shape

```

1. Graphics Class's Other Methods

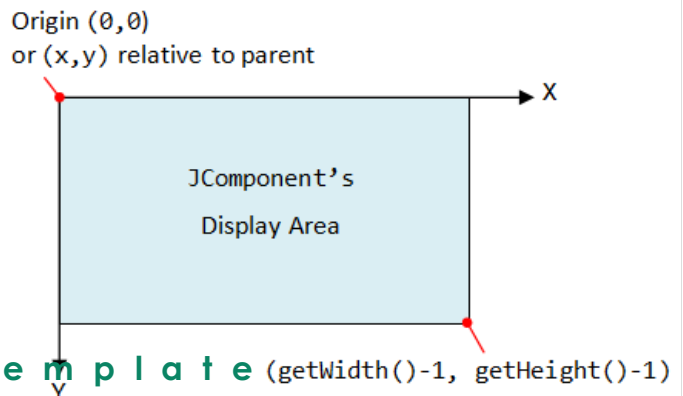
```
void clearRect(int x, int y, int width, int height)
    // Clear the rectangular area to background
void copyArea(int x, int y, int width, int height, int dx, int dy)
    // Copy the rectangular area to offset (dx, dy).
void translate(int x, int y)
    // Translate the origin of the graphics context to (x, y). Subsequent drawing uses the new origin.
FontMetrics getFontMetrics()
FontMetrics getFontMetrics(Font f)
    // Get the FontMetrics of the current font / the specified font
```

1.4 Graphics Coordinate System

In Java Windowing Subsystem (like most of the 2D Graphics systems), the origin (0,0) is located at the top-left corner.

EACH component/container has its own coordinate system, ranging from (0,0) to (width-1, height-1) as illustrated.

You can use method getWidth() and getHeight() to retrieve the width and height of a component/container. You can use getX() or getY() to get the top-left corner (x,y) of this component's origin relative to its parent.



2. Custom Painting Template (getWidth()-1, getHeight()-1)

Under Swing, custom painting is usually performed by extending (i.e., subclassing) a JPanel as the drawing canvas and override the paintComponent(Graphics g) method to perform your own drawing with the drawing methods provided by the Graphics class. The Java Windowing Subsystem invokes (calls back) paintComponent(g) to render the JPanel by providing the current graphics context g, which can be used to invoke the drawing methods. The extended JPanel is often programmed as an inner class of a JFrame application to facilitate access of private variables/methods. Although we typically draw on the JPanel, you can in fact draw on any JComponent (such as JLabel, JButton).

The custom painting code template is as follows:

```
1  import java.awt.*;
2  import javax.swing.*;
3
4  /** Custom Drawing Code Template */
5  @SuppressWarnings("serial")
6  public class CGTemplate extends JFrame { // Graphics application extends JFrame
7      // Named-constants for dimensions
8      public static final int CANVAS_WIDTH = 640;
9      public static final int CANVAS_HEIGHT = 480;
10
11      private DrawCanvas canvas; // Declare an instance the drawing canvas (extends JPanel)
12
13      /** Constructor to set up the GUI components */
14      public CGTemplate() {
15          canvas = new DrawCanvas(); // Construct the drawing canvas
16          canvas.setPreferredSize(new Dimension(CANVAS_WIDTH, CANVAS_HEIGHT));
17          this.setContentPane(canvas);
18          // Set the Drawing JPanel as the content-pane
19          // OR
20          // Get the JFrame's content-pane and add onto the content-pane as follows:
21          // Container cp = getContentPane();
22          // cp.add(canvas);
23
24          this.setDefaultCloseOperation(EXIT_ON_CLOSE); // Handle the CLOSE button
25          this.pack(); // Either pack() the components; or setSize()
26          this.setTitle("....."); // this JFrame sets the title
27          this.setVisible(true); // this JFrame show
28      }
29
30      /**
31       * DrawCanvas (inner class) is a JPanel used for custom drawing
32       */
33      private class DrawCanvas extends JPanel {
34          // Override paintComponent to perform your own painting
35          @Override
36          public void paintComponent(Graphics g) {
37              super.paintComponent(g); // paint parent's background
38              setBackground(Color.BLACK); // set background color for this JPanel
```

```

39
40     // Your custom painting codes. For example,
41     // Drawing primitive shapes
42     g.setColor(Color.YELLOW);    // set the drawing color
43     g.drawLine(30, 40, 100, 200);
44     g.drawOval(150, 180, 10, 10);
45     g.drawRect(200, 210, 20, 30);
46     g.setColor(Color.RED);      // change the drawing color
47     g.fillOval(300, 310, 30, 50);
48     g.fillRect(400, 350, 60, 50);
49     // Printing texts
50     g.setColor(Color.WHITE);
51     g.setFont(new Font("Courier New", Font.PLAIN, 12));
52     g.drawString("Testing custom drawing ...", 10, 20);
53 }
54 }
55
56 /** Entry main method */
57 public static void main(String[] args) {
58     // Run the GUI codes on the Event-Dispatching thread for thread safety
59     SwingUtilities.invokeLater(new Runnable() {
60         @Override
61         public void run() {
62             new CGTemplate(); // Let the constructor do the job
63         }
64     });
65 }
66 }

```

Dissecting the Program

- Custom painting is performed by extending a JPanel (called DrawCanvas) and overrides the paintComponent(Graphics g) method to do your own drawing with the drawing methods provided by the Graphics class.
- DrawCanvas is designed as an inner class of this JFrame application, so as to facilitate access of the private variables/methods.
- Java Windowing Subsystem invokes (calls back) paintComponent(g) to render the JPanel, with the current graphics context in g, whenever there is a need to refresh the display (e.g., during the initial launch, restore, resize, etc). You can use the drawing methods (g.drawXxx() and g.fillXxx()) on the current graphics context g to perform custom painting on the JPanel.
- The size of the JPanel is set via the setPreferredSize(). The JFrame does not set its size, but packs the components contained via pack().
- In the main(), the constructor is called in the event-dispatch thread via static method javax.swing.SwingUtilities.invokeLater() (instead of running in the main thread), to ensure thread-safety and avoid deadlock, as recommended by the Swing developers.

(Advanced) Anonymous Inner Class for Drawing Canvas

Instead of a named-inner class called DrawCanvas in the previous example, you can also use an anonymous inner class for the drawing canvas, if the painting code is short. For example,

```

// Create an anonymous inner class extends JPanel
// Construct an instance called canvas
JPanel canvas = new JPanel() {
    @Override
    public void paintComponent(Graphics g) {
        super.paintComponent(g); // paint parent's background
        .....
    }
};
.....

```

(Advanced) Getting the Graphics Context

You can retrieve the Graphics context of a JComponent via the getGraphics() method. This is, however, not commonly used. For example,

```

JPanel panel = new JPanel();
Graphics graphics = panel.getGraphics();

```

Custom Painting in AWT (Obsolete)

Under AWT, you can perform custom painting by extending java.awt.Canvas, and override the paint(Graphics g) method, in a java.awt.Frame application. Similarly, you can explicitly invoke repaint() to update the graphics.

2.1 Refreshing the Display via

At times, we need to *explicitly refresh the display* (e.g., in game and animation). We shall NOT invoke paintComponent(Graphics) directly. Instead, we invoke the JComponent's repaint() method. The Windowing Subsystem will in turn call back the paintComponent() with the current

Graphics context and execute it in the event-dispatching thread for thread safety. You can `repaint()` a particular `JComponent` (such as a `JPanel`) or the entire `JFrame`. The children contained within the `JComponent` will also be repainted.

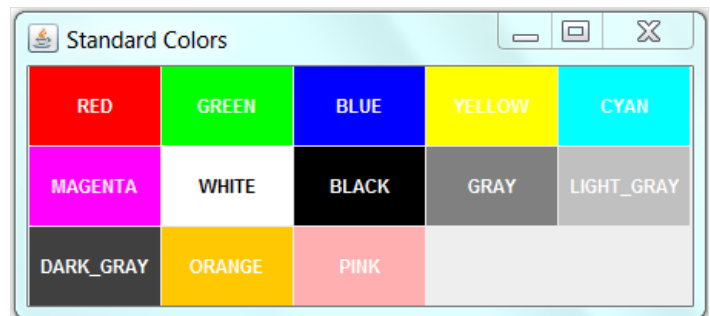
3. (Optional) Colors and Fonts

3. `java.awt.Color`

The class `java.awt.Color` provides 13 standard colors as named constants. They are: `Color.RED`, `GREEN`, `BLUE`, `MAGENTA`, `CYAN`, `YELLOW`, `BLACK`, `WHITE`, `GRAY`, `DARK_GRAY`, `LIGHT_GRAY`, `ORANGE`, and `PINK`. (In JDK 1.1, these constant names are in lowercase, e.g., `red`. This violates the Java naming convention for constants. In JDK 1.2, the uppercase names are added. The lowercase names were not removed for backward compatibility.)

You can use the `toString()` to print the RGB values of these color (e.g., `System.out.println(Color.RED)`):

```
RED      : java.awt.Color[r=255, g=0, b=0]
GREEN    : java.awt.Color[r=0, g=255, b=0]
BLUE     : java.awt.Color[r=0, g=0, b=255]
YELLOW   : java.awt.Color[r=255, g=255, b=0]
MAGENTA  : java.awt.Color[r=255, g=0, b=255]
CYAN     : java.awt.Color[r=0, g=255, b=255]
WHITE    : java.awt.Color[r=255, g=255, b=255]
BLACK    : java.awt.Color[r=0, g=0, b=0]
GRAY     : java.awt.Color[r=128, g=128, b=128]
LIGHT_GRAY: java.awt.Color[r=192, g=192, b=192]
DARK_GRAY: java.awt.Color[r=64, g=64, b=64]
PINK     : java.awt.Color[r=255, g=175, b=175]
ORANGE   : java.awt.Color[r=255, g=200, b=0]
```



You can also use the RGB values or RGBA value (A for alpha to specify transparency/opaque) to construct your own color via constructors:

```
Color(int r, int g, int b);           // between 0 and 255
Color(float r, float g, float b);     // between 0.0f and 1.0f
Color(int r, int g, int b, int alpha); // between 0 and 255
Color(float r, float g, float b, float alpha); // between 0.0f and 1.0f
// alpha of 0 for totally transparent, 255 (or 1.0f) for totally opaque
// The default alpha is 255 (or 1.0f) for totally opaque
```

For example:

```
Color myColor1 = new Color(123, 111, 222);
Color myColor2 = new Color(0.5f, 0.3f, 0.1f);
Color myColor3 = new Color(0.5f, 0.3f, 0.1f, 0.5f); // semi-transparent
```

To retrieve the individual components, you can use `getRed()`, `getGreen()`, `getBlue()`, `getAlpha()`, etc.

To set the background and foreground (text) color of a component/container, you can invoke:

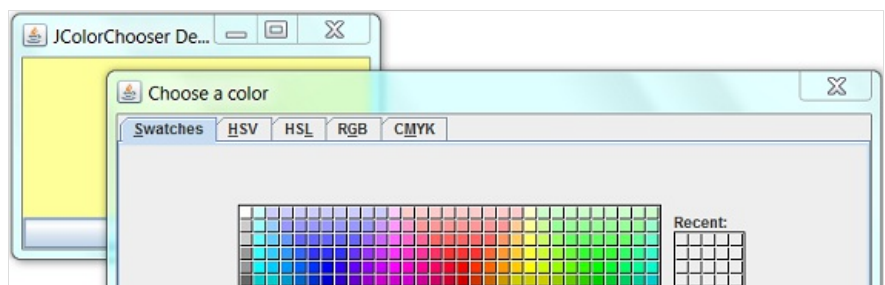
```
JLabel label = new JLabel("Test");
label.setBackground(Color.LIGHT_GRAY);
label.setForeground(Color.RED);
```

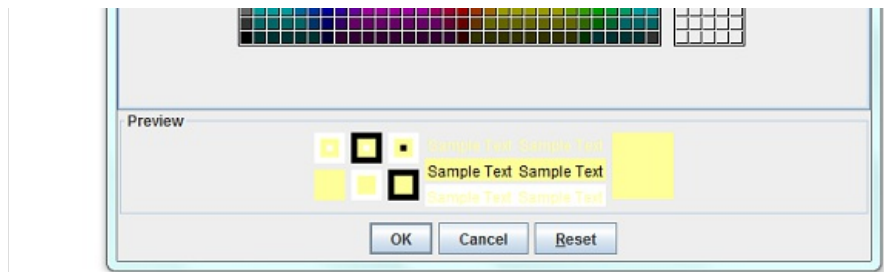
To set the color of the Graphics context `g` (for drawing lines, shapes, and texts), use `g.setColor(color)`:

```
g.setColor(Color.RED);
g.drawLine(10, 20, 30, 40); // in Color.RED
Color myColor = new Color(123, 111, 222);
g.setColor(myColor);
g.drawRect(10, 10, 40, 50); // in myColor
```

(Advancing) Color Chooser

This example uses the `javax.swing.JColorChooser` to set the background color of the `JPanel`.





```

1  import java.awt.*;
2  import java.awt.event.*;
3  import javax.swing.*;
4
5  /** Test ColorChooser to set the background */
6  @SuppressWarnings("serial")
7  public class JColorChooserDemo extends JFrame {
8
9      JPanel panel;
10     Color bgColor = Color.LIGHT_GRAY; // panel's background color
11
12     /** Constructor to setup the UI components */
13     public JColorChooserDemo() {
14         panel = new JPanel(new BorderLayout());
15
16         JButton btnColor = new JButton("Change Color");
17         panel.add(btnColor, BorderLayout.SOUTH);
18         btnColor.addActionListener(new ActionListener() {
19             @Override
20             public void actionPerformed(ActionEvent e) {
21                 Color color = JColorChooser.showDialog(JColorChooserDemo.this,
22                     "Choose a color", bgColor);
23                 if (color != null) { // new color selected
24                     bgColor = color;
25                 }
26                 panel.setBackground(bgColor); // change panel's background color
27             }
28         });
29
30         setContentPane(panel);
31
32         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
33         setTitle("JColorChooser Demo");
34         setSize(300, 200);
35         setLocationRelativeTo(null); // center the application window
36         setVisible(true);           // show it
37     }
38
39     /** The entry main() method */
40     public static void main(String[] args) {
41         // Run GUI codes in the Event-Dispatching thread for thread safety
42         SwingUtilities.invokeLater(new Runnable() {
43             @Override
44             public void run() {
45                 new JColorChooserDemo(); // Let the constructor do the job
46             }
47         });
48     }
49 }

```

3.3 java.awt.Font

The class `java.awt.Font` represents a specific font face, which can be used for rendering texts. You can use the following constructor to construct a `Font` instance:

```

public Font(String name, int style, int size);
// name: Family name "Dialog", "DialogInput", "Monospaced", "Serif", or "SansSerif" or
//       Physical font found in this GraphicsEnvironment.
//       You can also use String constants Font.DIALOG, Font.DIALOG_INPUT, Font.MONOSPACED,
//       Font.SERIF, Font.SANS_SERIF (JDK 1.6)
// style: Font.PLAIN, Font.BOLD, Font.ITALIC or Font.BOLD|Font.ITALIC (Bit-OR)
// size: the point size of the font (in pt) (1 inch has 72 pt).

```

You can use the `setFont()` method to set the current font for the `Graphics` context `g` for rendering texts. For example,

```

Font myFont1 = new Font(Font.MONOSPACED, Font.PLAIN, 12);
Font myFont2 = new Font(Font.SERIF, Font.BOLD | Font.ITALIC, 16); // bold and italics

```

```

JButton btn = new JButton("RESET");
btn.setFont(myFont1);
JLabel lbl = new JLabel("Hello");
lbl.setFont(myFont2);
.....
g.drawString("In default Font", 10, 20);    // in default font
Font myFont3 = new Font(Font.SANS_SERIF, Font.ITALIC, 12);
g.setFont(myFont3);
g.drawString("Using the font set", 10, 50); // in myFont3

```

Font's Family Name vs. Font Name

A font could have many *faces* (or *style*), e.g., plain, bold or italic. All these faces have similar typographic design. The *font face name*, or *font name* for short, is the name of a particular font face, like "Arial", "Arial Bold", "Arial Italic", "Arial Bold Italic". The *font family name* is the name of the font family that determines the typographic design across several faces, like "Arial". For example,

```

java.awt.Font[family=Arial,name=Arial,style=plain,size=1]
java.awt.Font[family=Arial,name=Arial Bold,style=plain,size=1]
java.awt.Font[family=Arial,name=Arial Bold Italic,style=plain,size=1]
java.awt.Font[family=Arial,name=Arial Italic,style=plain,size=1]

```

Logical Font vs. Physical Font

JDK supports these *logical font family* names: "Dialog", "DialogInput", "Monospaced", "Serif", or "SansSerif". JDK 1.6 provides these String constants: Font.DIALOG, Font.DIALOG_INPUT, Font.MONOSPACED, Font.SERIF, Font.SANS_SERIF.

Physical font names are actual font libraries such as "Arial", "Times New Roman" in the system.

GraphicsEnvironment.getAvailableFontFamilyNames() and GraphicsEnvironment.getAllFonts()

You can use GraphicsEnvironment's getAvailableFontFamilyNames() to list all the font family names; and getAllFonts() to construct all Font instances (with font size of 1 pt). For example,

```

GraphicsEnvironment env = GraphicsEnvironment.getLocalGraphicsEnvironment();

// Get all font family name in a String[]
String[] fontNames = env.getAvailableFontFamilyNames();
for (String fontName : fontNames) {
    System.out.println(fontName);
}

// Construct all Font instance (with font size of 1)
Font[] fonts = env.getAllFonts();
for (Font font : fonts) {
    System.out.println(font);
}

```

Font.deriveFont()

You can use Font's deriveFont() to derive a new Font instance from this Font with varying size, style and others.

```

public Font deriveFont(float size)
public Font deriveFont(int style)
public Font deriveFont(AffineTransform trans)
public Font deriveFont(int style, float size)
public Font deriveFont(int style, AffineTransform trans)

```

For example,

```

Font font = new Font(Font.MONOSPACED, Font.BOLD, 12);
System.out.println(font);
// java.awt.Font[family=Monospaced,name=Monospaced,style=bold,size=12]
Font fontDerived = font.deriveFont(20);
System.out.println(fontDerived);
// java.awt.Font[family=Monospaced,name=Monospaced,style=plain,size=12]

```

3.3 java.awt.FontMetrics

The java.awt.FontMetrics class can be used to measure the exact width and height of the string for a particular font face, so that you can position the string as you desire (such as at the center of the screen).

To create a FontMetrics, use getFontMetrics() methods of the Graphics class, as follows:

```

// In java.awt.Graphics
public abstract FontMetrics getFontMetrics(Font f)
// Get the FontMetrics of the specified font
public abstract FontMetrics getFontMetrics()

```



```
// Get the FontMetrics of the current font
```



```
// in java.awt.FontMetrics
public int getHeight()
public int getLeading()
public int getAscent()
public int getDescent()
```

The most commonly-used function for FontMetrics is to measure the width of a given String displayed in a certain font.

```
public int stringWidth(String str)
// Returns the total width for showing the specified String in this Font.
```

To centralize a string on the drawing canvas (e.g., JPanel):

```
public void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.setFont(new Font("Arial", Font.BOLD, 30));
    // Get font metrics for the current font
    FontMetrics fm = g.getFontMetrics();
    // Centralize the string
    String msg = "Hello, world!";
    int msgWidth = fm.stringWidth(msg);
    int msgAscent = fm.getAscent();
    // Get the position of the leftmost character in the baseline
    // getWidth() and getHeight() returns the width and height of this component
    int msgX = getWidth() / 2 - msgWidth / 2;
    int msgY = getHeight() / 2 + msgAscent / 2;
    g.drawString(msg, msgX, msgY);
}
```

4. Example 1: Moving an Object via Key / Button

This example illustrates how to re-paint the screen in response to a KeyEvent or ActionEvent.

The display consists of two JPanel in a JFrame, arranged in BorderLayout. The top panel is used for custom painting; the bottom panel holds two JButton arranged in FlowLayout. Clicking the "Move Right" or "Move Left" buttons moves the line. The JFrame listens to the "Left-arrow" and "Right-arrow" keys, and responds by moving the line left or right.



```
1  import java.awt.*;
2  import java.awt.event.*;
3  import javax.swing.*;
4  /**
5   * Custom Graphics Example: Using key/button to move a line left or right.
6   */
7  @SuppressWarnings("serial")
8  public class CGMoveALine extends JFrame {
9      // Name-constants for the various dimensions
10     public static final int CANVAS_WIDTH = 400;
11     public static final int CANVAS_HEIGHT = 140;
12     public static final Color LINE_COLOR = Color.BLACK;
13     public static final Color CANVAS_BACKGROUND = Color.CYAN;
14
15     // The line from (x1, y1) to (x2, y2), initially position at the center
16     private int x1 = CANVAS_WIDTH / 2;
17     private int y1 = CANVAS_HEIGHT / 8;
18     private int x2 = x1;
19     private int y2 = CANVAS_HEIGHT / 8 * 7;
20
21     private DrawCanvas canvas; // the custom drawing canvas (extends JPanel)
22
23     /** Constructor to set up the GUI */
24     public CGMoveALine() {
25         // Set up a panel for the buttons
```



```

26     JPanel btnPanel = new JPanel(new FlowLayout());
27     JButton btnLeft = new JButton("Move Left ");
28     btnPanel.add(btnLeft);
29     btnLeft.addActionListener(new ActionListener() {
30         public void actionPerformed(ActionEvent e) {
31             x1 -= 10;
32             x2 -= 10;
33             canvas.repaint();
34             requestFocus(); // change the focus to JFrame to receive KeyEvent
35         }
36     });
37     JButton btnRight = new JButton("Move Right");
38     btnPanel.add(btnRight);
39     btnRight.addActionListener(new ActionListener() {
40         public void actionPerformed(ActionEvent e) {
41             x1 += 10;
42             x2 += 10;
43             canvas.repaint();
44             requestFocus(); // change the focus to JFrame to receive KeyEvent
45         }
46     });
47
48     // Set up a custom drawing JPanel
49     canvas = new DrawCanvas();
50     canvas.setPreferredSize(new Dimension(CANVAS_WIDTH, CANVAS_HEIGHT));
51
52     // Add both panels to this JFrame
53     Container cp = getContentPane();
54     cp.setLayout(new BorderLayout());
55     cp.add(canvas, BorderLayout.CENTER);
56     cp.add(btnPanel, BorderLayout.SOUTH);
57
58     // "this" JFrame fires KeyEvent
59     addKeyListener(new KeyAdapter() {
60         @Override
61         public void keyPressed(KeyEvent evt) {
62             switch(evt.getKeyCode()) {
63                 case KeyEvent.VK_LEFT:
64                     x1 -= 10;
65                     x2 -= 10;
66                     repaint();
67                     break;
68                 case KeyEvent.VK_RIGHT:
69                     x1 += 10;
70                     x2 += 10;
71                     repaint();
72                     break;
73             }
74         }
75     });
76
77     setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); // Handle the CLOSE button
78     setTitle("Move a Line");
79     pack(); // pack all the components in the JFrame
80     setVisible(true); // show it
81     requestFocus(); // set the focus to JFrame to receive KeyEvent
82 }
83
84 /**
85  * DrawCanvas (inner class) is a JPanel used for custom drawing
86  */
87 class DrawCanvas extends JPanel {
88     @Override
89     public void paintComponent(Graphics g) {
90         super.paintComponent(g);
91         setBackground(CANVAS_BACKGROUND);
92         g.setColor(LINE_COLOR);
93         g.drawLine(x1, y1, x2, y2); // draw the line
94     }
95 }
96
97 /** The entry main() method */
98 public static void main(String[] args) {
99     // Run GUI codes on the Event-Dispatcher Thread for thread safety
100    SwingUtilities.invokeLater(new Runnable() {
101        @Override
102        public void run() {
103            new CGMoveALine(); // Let the constructor do the job
104        }
105    });

```

```

106     }
107 }

```

Dissecting the Program

- To do custom painting, you have to decide which superclass to use. It is recommended that you use a JPanel (or a more specialized Swing component such as JButton or JLabel). In this example, we extend the JPanel to do our custom painting, in an inner class, as follows:

```

class DrawCanvas extends JPanel {
    @Override
    public void paintComponent(Graphics g) {
        super.paintComponent(g); // paint background
        setBackground(CANVAS_BACKGROUND);
        g.setColor(LINE_COLOR);
        g.drawLine(x1, y1, x2, y2);
    }
}

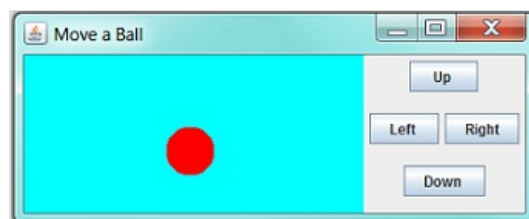
```

- The paintComponent() method is overridden to provide the custom drawing codes. We use the drawLine() method to draw a line from (x1,y1) to (x2, y2).
- The paintComponent() method cannot be called directly from your code, because it requires a Graphics object as argument.
- paintComponent() is a so-called "call-back" method. The Windowing subsystem invokes this method and provides a pre-configured Graphics object to represent its state (e.g., current color, font, clip area and etc). There are two kinds of painting: system-triggered painting and application-triggered painting. In a system-trigger painting, the system request a component to render its content when the component is first made visible on the screen, or the component is resized, or the component is damaged that needs to be repaint. In an application-triggered painting, the application invokes a repaint() request. Under both cases, the Windowing subsystem will call-back the paintComponent() to render the contents of the component with a proper Graphics object as argument.
- In this example, the application requests for a repaint() in the KeyEvent and MouseEvent handlers, which triggers the paintComponent() with an appropriate Graphics object as the argument.
- To be precise, when you invoke the repaint() method to repaint a JComponent, the Windowing subsystem calls-back paint() method. The paint() method then calls-back three methods: paintComponent(), paintBorder() and paintChildren().
- In the overridden paintComponent() method, we call super.paintComponent() to paint the background of the JComponent. If this call is omitted, you must either paint the background yourself (via a fillRect() call) or use setOpaque(false) to make the JComponent transparent. This will inform Swing system to paint those JComponents behind the transparent component.
- We choose the JFrame as the source of the KeyEvent. JFrame shall be "in focus" when the key is pressed. The requestFocus() method (of "this" JFrame) is invoked to request for the keyboard focus.

[TODO]: may need to revise.

Try

Modifying the program to move a ball in response to up/down/left/right buttons, as well as the 4 arrow and "wasd" keys, as shown:



5. Example 2: Moving Sprites

In game programming, we have moving game objects called *sprites*. Each sprite is usually modelled in its own class, with its own properties, and it can paint itself.

Sprite.java

This class models a sprite, with its own properties, and it can paint itself via the paint() method provided given a Graphics context. A rectangle is used here.

```

1  import java.awt.*;
2  /**
3   * The class Srite models a moving game object, with its own operations
4   * and can paint itself.
5   */
6  public class Sprite {
7      // Variables (package access)
8      int x, y, width, height; // rectangle (for illustration)
9      Color color = Color.RED; // color of the object
10

```

```

11  /** Constructor to setup the GUI */
12  public Sprite(int x, int y, int width, int height, Color color) {
13      this.x = x;
14      this.y = y;
15      this.width = width;
16      this.height = height;
17      this.color = color;
18  }
19
20  /** Paint itself (given the Graphics context) */
21  public void paint(Graphics g) {
22      g.setColor(color);
23      g.fillRect(x, y, width, height); // fill a rectangle
24  }
25  }

```

MoveASprite.java

Instead of repainting the entire display, we only repaint the affected areas (clips), for efficiency, via the `repaint(x, y, width, height)` method. In `moveLeft()` and `moveRight()`, we save the states, move the object, repaint the saved clip-area with the background color, and repaint the new clip-area occupied by the sprite. Repainting is done by asking the sprite to paint itself at the new location, and erase from the old location.

```

1  import java.awt.*;
2  import java.awt.event.*;
3  import javax.swing.*;
4  /**
5   * Custom Graphics Example: Using key/button to move a object left or right.
6   * The moving object (sprite) is defined in its own class, with its own
7   * operations and can paint itself.
8   */
9  public class CGMoveASprite extends JFrame {
10     // Name-constants for the various dimensions
11     public static final int CANVAS_WIDTH = 400;
12     public static final int CANVAS_HEIGHT = 140;
13     public static final Color CANVAS_BG_COLOR = Color.CYAN;
14
15     private DrawCanvas canvas; // the custom drawing canvas (extends JPanel)
16     private Sprite sprite;     // the moving object
17
18     /** Constructor to set up the GUI */
19     public CGMoveASprite() {
20         // Construct a sprite given x, y, width, height, color
21         sprite = new Sprite(CANVAS_WIDTH / 2 - 5, CANVAS_HEIGHT / 2 - 40,
22                             10, 80, Color.RED);
23
24         // Set up a panel for the buttons
25         JPanel btnPanel = new JPanel(new FlowLayout());
26         JButton btnLeft = new JButton("Move Left ");
27         btnPanel.add(btnLeft);
28         btnLeft.addActionListener(new ActionListener() {
29             @Override
30             public void actionPerformed(ActionEvent e) {
31                 moveLeft();
32                 requestFocus(); // change the focus to JFrame to receive KeyEvent
33             }
34         });
35         JButton btnRight = new JButton("Move Right");
36         btnPanel.add(btnRight);
37         btnRight.addActionListener(new ActionListener() {
38             @Override
39             public void actionPerformed(ActionEvent e) {
40                 moveRight();
41                 requestFocus(); // change the focus to JFrame to receive KeyEvent
42             }
43         });
44
45         // Set up the custom drawing canvas (JPanel)
46         canvas = new DrawCanvas();
47         canvas.setPreferredSize(new Dimension(CANVAS_WIDTH, CANVAS_HEIGHT));
48
49         // Add both panels to this JFrame
50         Container cp = getContentPane();
51         cp.setLayout(new BorderLayout());
52         cp.add(canvas, BorderLayout.CENTER);
53         cp.add(btnPanel, BorderLayout.SOUTH);
54
55         // "this" JFrame fires KeyEvent
56         addKeyListener(new KeyAdapter() {
57             @Override

```

```

58     public void keyPressed(KeyEvent evt) {
59         switch(evt.getKeyCode()) {
60             case KeyEvent.VK_LEFT: moveLeft(); break;
61             case KeyEvent.VK_RIGHT: moveRight(); break;
62         }
63     }
64 });
65
66     setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
67     setTitle("Move a Sprite");
68     pack(); // pack all the components in the JFrame
69     setVisible(true); // show it
70     requestFocus(); // "this" JFrame requests focus to receive KeyEvent
71 }
72
73 /** Helper method to move the sprite left */
74 private void moveLeft() {
75     // Save the current dimensions for repaint to erase the sprite
76     int savedX = sprite.x;
77     // update sprite
78     sprite.x -= 10;
79     // Repaint only the affected areas, not the entire JFrame, for efficiency
80     canvas.repaint(savedX, sprite.y, sprite.width, sprite.height); // Clear old area to background
81     canvas.repaint(sprite.x, sprite.y, sprite.width, sprite.height); // Paint new location
82 }
83
84 /** Helper method to move the sprite right */
85 private void moveRight() {
86     // Save the current dimensions for repaint to erase the sprite
87     int savedX = sprite.x;
88     // update sprite
89     sprite.x += 10;
90     // Repaint only the affected areas, not the entire JFrame, for efficiency
91     canvas.repaint(savedX, sprite.y, sprite.width, sprite.height); // Clear old area to background
92     canvas.repaint(sprite.x, sprite.y, sprite.width, sprite.height); // Paint at new location
93 }
94
95 /** DrawCanvas (inner class) is a JPanel used for custom drawing */
96 class DrawCanvas extends JPanel {
97     @Override
98     public void paintComponent(Graphics g) {
99         super.paintComponent(g);
100         setBackground(CANVAS_BG_COLOR);
101         sprite.paint(g); // the sprite paints itself
102     }
103 }
104
105 /** The entry main() method */
106 public static void main(String[] args) {
107     // Run GUI construction on the Event-Dispatching Thread for thread safety
108     SwingUtilities.invokeLater(new Runnable() {
109         @Override
110         public void run() {
111             new CGMoveASprite(); // Let the constructor do the job
112         }
113     });
114 }
115 }

```

6. Example 3: Paint



MyPaint.java

```

1  import java.util.List;
2  import java.util.ArrayList;
3  import javax.swing.*;
4  import java.awt.*;
5  import java.awt.event.*;
6
7  /**
8   * Custom Graphics Example: Paint

```

```

9  */
10 public class MyPaint extends JFrame {
11     // Name-constants for the various dimensions
12     public static final int CANVAS_WIDTH = 500;
13     public static final int CANVAS_HEIGHT = 300;
14     public static final Color LINE_COLOR = Color.RED;
15
16     // Lines, consists of a List of PolyLine instances
17     private List<PolyLine> lines = new ArrayList<PolyLine>();
18     private PolyLine currentLine; // the current line (for capturing)
19
20     /** Constructor to set up the GUI */
21     public MyPaint() {
22         DrawCanvas canvas = new DrawCanvas();
23         canvas.setPreferredSize(new Dimension(CANVAS_WIDTH, CANVAS_HEIGHT));
24         canvas.addMouseListener(new MouseAdapter() {
25             @Override
26             public void mousePressed(MouseEvent e) {
27                 // Begin a new line
28                 currentLine = new PolyLine();
29                 lines.add(currentLine);
30                 currentLine.addPoint(e.getX(), e.getY());
31             }
32         });
33         canvas.addMouseMotionListener(new MouseMotionAdapter() {
34             @Override
35             public void mouseDragged(MouseEvent e) {
36                 currentLine.addPoint(e.getX(), e.getY());
37                 repaint(); // invoke paintComponent()
38             }
39         });
40
41         setContentPane(canvas);
42         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
43         setTitle("Paint");
44         pack();
45         setVisible(true);
46     }
47
48     /** DrawCanvas (inner class) is a JPanel used for custom drawing */
49     private class DrawCanvas extends JPanel {
50         @Override
51         protected void paintComponent(Graphics g) { // called back via repaint()
52             super.paintComponent(g);
53             g.setColor(LINE_COLOR);
54             for (PolyLine line: lines) {
55                 line.draw(g);
56             }
57         }
58     }
59
60     /** The entry main method */
61     public static void main(String[] args) {
62         SwingUtilities.invokeLater(new Runnable() {
63             // Run the GUI codes on the Event-Dispatching thread for thread safety
64             @Override
65             public void run() {
66                 new MyPaint(); // Let the constructor do the job
67             }
68         });
69     }
70 }

```

P o l y L i n e . j a v a

```

1  import java.awt.Graphics;
2  import java.util.*;
3  /*
4   * The PolyLine class model a line consisting of many points
5   */
6  public class PolyLine {
7      private List<Integer> xList; // List of x-coord
8      private List<Integer> yList; // List of y-coord
9
10     /** Constructor */
11     public PolyLine() {
12         xList = new ArrayList<Integer>();
13         yList = new ArrayList<Integer>();
14     }
15 }

```

```

16  /** Add a point to this PolyLine */
17  public void addPoint(int x, int y) {
18      xList.add(x);
19      yList.add(y);
20  }
21
22  /** This PolyLine paints itself */
23  public void draw(Graphics g) { // draw itself
24      for (int i = 0; i < xList.size() - 1; ++i) {
25          g.drawLine((int)xList.get(i), (int)yList.get(i), (int)xList.get(i + 1),
26                    (int)yList.get(i + 1));
27      }
28  }
29  }

```

Dissecting the Program

[TODO]

7. Drawing Images

7.1 `javax.swing.ImageIcon`

The `javax.swing.ImageIcon` class represents an icon, which is a fixed-size picture, typically small-size and used to decorate components. To create an `ImageIcon`:

```

// Prepare an ImageIcon to be used with JComponents or drawImage()
String imgNoughtFilename = "images/nought.gif";
ImageIcon iconNought = null;
URL imgURL = getClass().getClassLoader().getResource(imgNoughtFilename);
if (imgURL != null) {
    iconNought = new ImageIcon(imgURL);
} else {
    System.err.println("Couldn't find file: " + imgNoughtFilename);
}

```

7.2 `Graphics.drawImage()`

`ImageIcon` is fixed-in-sized and cannot be resized in display. You can use `Graphics`'s `drawImage()` to resize a source image in display.

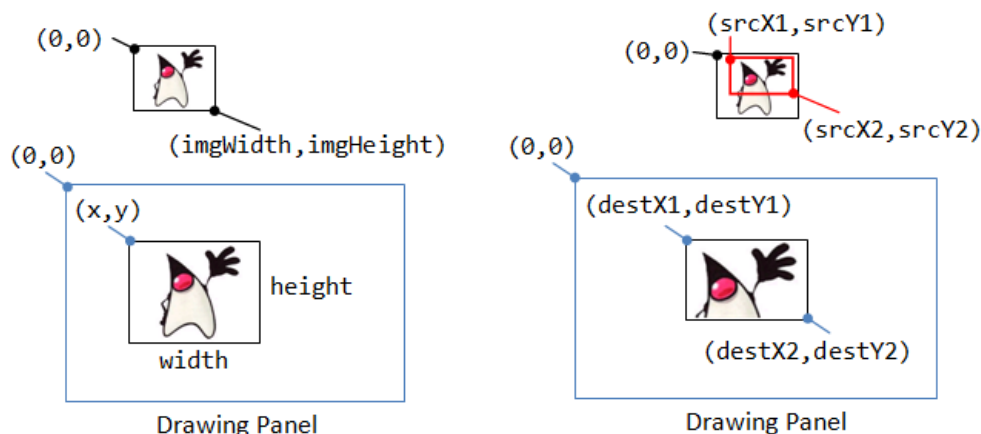
The `java.awt.Graphics` class declares 6 overloaded versions of abstract method `drawImage()`.

```

public abstract boolean drawImage(Image img, int x, int y, ImageObserver observer)
public abstract boolean drawImage(Image img, int x, int y, int width, int height, ImageObserver observer)
public abstract boolean drawImage(Image img, int x, int y, Color bgColor, ImageObserver observer)
public abstract boolean drawImage(Image img, int x, int y, int width, int height, Color bgColor, ImageObserver observer)
    // The img is drawn with its top-left corner at (x, y) scaled to the specified width and height
    // (default to the image's width and height).
    // The bgColor (background color) is used for "transparent" pixels.

public abstract boolean drawImage(Image img, int destX1, int destY1, int destX2, int destY2,
    int srcX1, int srcY1, int srcX2, int srcY2, ImageObserver observer)
public abstract boolean drawImage(Image img, int destX1, int destY1, int destX2, int destY2,
    int srcX1, int srcY1, int srcX2, int srcY2, Color bgColor, ImageObserver observer)
    // The img "clip" bounded by (srcX1, srcY2) and (srcX2, srcY2) is scaled and drawn from
    // (destX1, destY1) to (destX2, destY2) on the display.

```



The coordinates involved is shown in the above diagram. The `ImageObserver` receives notification about the `Image` as it is loaded. In most

purposes, you can set it to null or this.

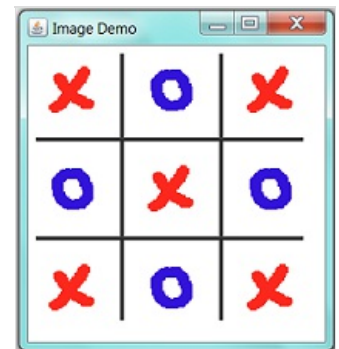
The `drawImage()` method requires an `Image` instance, which can be obtained via `ImageIcon`'s `getImage()` method; or via static method `ImageIO.read()` (read "[Reading Images into your program](#)"). For example,

```
// Prepare an ImageIcon
ImageIcon icon = null;
String imgFilename = "images/duke.gif";
java.net.URL imgURL = getClass().getClassLoader().getResource(imgFilename);
if (imgURL != null) {
    icon = new ImageIcon(imgURL);
} else {
    System.err.println("Couldn't find file: " + imgFilename);
}

// Prepare an Image object to be used by drawImage()
final Image img = icon.getImage();
// Extend a JLabel and override paintComponent() to drawImage()
JLabel lbl4 = new JLabel() {
    @Override public void paintComponent(Graphics g) {
        super.paintComponent(g); // paint background
        g.drawImage(img, 0, 0, 200, 200, null);
    }
};
lbl4.setPreferredSize(new Dimension(200, 200));
cp.add(lbl4);
```

Example

Images:



```
1 import java.awt.*;
2 import java.net.URL;
3 import javax.swing.*;
4 import java.util.Random;
5
6 /** Test drawImage() thru ImageIcon */
7 @SuppressWarnings("serial")
8 public class CGDrawImageDemo extends JFrame {
9     // Define named-constants for the various dimensions
10     public static final int ROWS = 3;
11     public static final int COLS = 3;
12     public static final int IMAGE_SIZE = 50;
13     public static final int PADDING = 20; // padding from the border
14     public static final int CELL_SIZE = IMAGE_SIZE + 2 * PADDING;
15     public static final int CANVAS_SIZE = CELL_SIZE * ROWS;
16
17     private DrawCanvas canvas; // The drawing canvas
18     private Random random = new Random(); // for picking images in random
19
20     // Images
21     private String imgCrossFilename = "images/cross.gif";
22     private String imgNoughtFilename = "images/nought.gif";
23     private Image imgCross; // drawImage() uses an Image object
24     private Image imgNought;
25
26     /** Constructor to set up the GUI components */
27     public CGDrawImageDemo() {
28         // Prepare the ImageIcon and Image objects for drawImage()
29         ImageIcon iconCross = null;
30         ImageIcon iconNought = null;
31         URL imgURL = getClass().getClassLoader().getResource(imgCrossFilename);
32         if (imgURL != null) {
33             iconCross = new ImageIcon(imgURL);
34         } else {
35             System.err.println("Couldn't find file: " + imgCrossFilename);
36         }
37     }
38 }
```



```

37     imgCross = iconCross.getImage();
38
39     imgURL = getClass().getClassLoader().getResource(imgNoughtFilename);
40     if (imgURL != null) {
41         iconNought = new ImageIcon(imgURL);
42     } else {
43         System.err.println("Couldn't find file: " + imgNoughtFilename);
44     }
45     imgNought = iconNought.getImage();
46
47     canvas = new DrawCanvas();
48     canvas.setPreferredSize(new Dimension(CANVAS_SIZE, CANVAS_SIZE));
49     this.setContentPane(canvas); // use JPanel as content-pane
50     this.setDefaultCloseOperation(EXIT_ON_CLOSE);
51     this.pack(); // pack the components of this JFrame
52     this.setTitle("Test drawImage()");
53     this.setVisible(true);
54 }
55
56 /** DrawCanvas (inner class) is a JPanel used for custom drawing */
57 private class DrawCanvas extends JPanel {
58     @Override
59     public void paintComponent(Graphics g) {
60         super.paintComponent(g);
61         setBackground(Color.WHITE); // Set background color for this JPanel
62         // Drawing Images (picked in random)
63         for (int row = 0; row < ROWS; ++row) {
64             for (int col = 0; col < COLS; ++col) {
65                 boolean useCross = random.nextBoolean();
66                 Image img = useCross ? imgCross : imgNought;
67                 g.drawImage(img,
68                     CELL_SIZE * col + PADDING, CELL_SIZE * row + PADDING,
69                     IMAGE_SIZE, IMAGE_SIZE, null);
70             }
71         }
72         // Draw Borders
73         g.fillRect(CELL_SIZE - 2, 0, 4, CELL_SIZE * 3, true);
74         g.fillRect(CELL_SIZE * 2 - 2, 0, 4, CELL_SIZE * 3, true);
75         g.fillRect(0, CELL_SIZE - 2, CELL_SIZE * 3, 4, true);
76         g.fillRect(0, CELL_SIZE * 2 - 2, CELL_SIZE * 3, 4, true);
77     }
78 }
79
80 /** The entry main method */
81 public static void main(String[] args) {
82     // Run the GUI codes on the Event-Dispatching thread for thread-safety
83     SwingUtilities.invokeLater(new Runnable() {
84         @Override
85         public void run() {
86             new CGDrawImageDemo(); // Let the constructor do the job
87         }
88     });
89 }
90 }

```

This example places absolute numbers in the draw methods, which is hard to maintain and reuse. You should define name-constants such as `CELL_WIDTH`, `BORDER_WIDTH`, etc, and compute the numbers based on these constants.

8 . A n i m a t i o n

8 . 1 A n i m a t i o n u s i n g . T i m e r

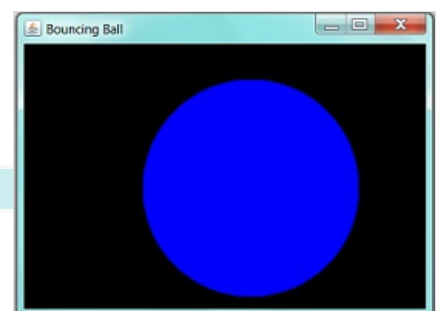
Creating an animation (such as a bouncing ball) requires repeatedly running an updating task at a regular interval. Swing provides a `javax.swing.Timer` class which can be used to fire `ActionEvent` to its registered `ActionListeners` at regular interval.

The `Timer` class has one constructor:

```
public Timer(int delay, ActionListener listener)
```

You are required to override the `actionPerformed()` method of the `ActionListener` to specify your task's behavior. The `Timer` fires an `ActionEvent` to the `ActionListener` after the (initial) delay, and then at regular interval after delay.

You can start and stop the `Timer` via the `Timer`'s `start()` and `stop()` methods. For example,



```

int delay = 500; // milliseconds
// Create an instance of an anonymous subclass of ActionListener
ActionListener updateTask = new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent evt) {
        // .....
    }
};
// Start and run the task at regular delay
new Timer(delay, updateTask).start();

```

You can use method `setRepeats(false)` to set the `Timer` to fire only once, after the delay. You can set the initial delay via `setInitialDelay()` and regular delay via `setDelay()`.

A `Timer` can fire the `ActionEvent` to more than one `ActionListeners`. You can register more `ActionListeners` via the `addActionListener()` method.

The `actionPerformed()` runs on the event-dispatching thread, just like all the event handlers. You can be relieved of the multi-threading issues.

JDK 1.3 introduced another timer class called `java.util.Timer`, which is more general, but `javax.swing.Timer` is sufficient (and easier) to run animation in Swing application.

Example: A Bouncing Ball

```

1  import java.awt.*;
2  import java.awt.event.*;
3  import javax.swing.*;
4
5  /** Bouncing Ball (Animation) via Swing Timer */
6  @SuppressWarnings("serial")
7  public class CGBouncingBallSwingTimer extends JFrame {
8      // Define named-constants
9      private static final int CANVAS_WIDTH = 640;
10     private static final int CANVAS_HEIGHT = 480;
11     private static final int UPDATE_PERIOD = 50; // milliseconds
12
13     private DrawCanvas canvas; // the drawing canvas (extends JPanel)
14
15     // Attributes of moving object
16     private int x = 100, y = 100; // top-left (x, y)
17     private int size = 250; // width and height
18     private int xSpeed = 3, ySpeed = 5; // displacement per step in x, y
19
20     /** Constructor to setup the GUI components */
21     public CGBouncingBallSwingTimer() {
22         canvas = new DrawCanvas();
23         canvas.setPreferredSize(new Dimension(CANVAS_WIDTH, CANVAS_HEIGHT));
24         this.setContentPane(canvas);
25         this.setDefaultCloseOperation(EXIT_ON_CLOSE);
26         this.pack();
27         this.setTitle("Bouncing Ball");
28         this.setVisible(true);
29
30         // Define an ActionListener to perform update at regular interval
31         ActionListener updateTask = new ActionListener() {
32             @Override
33             public void actionPerformed(ActionEvent evt) {
34                 update(); // update the (x, y) position
35                 repaint(); // Refresh the JFrame, callback paintComponent()
36             }
37         };
38         // Allocate a Timer to run updateTask's actionPerformed() after every delay msec
39         new Timer(UPDATE_PERIOD, updateTask).start();
40     }
41
42     /** Update the (x, y) position of the moving object */
43     public void update() {
44         x += xSpeed;
45         y += ySpeed;
46         if (x > CANVAS_WIDTH - size || x < 0) {
47             xSpeed = -xSpeed;
48         }
49         if (y > CANVAS_HEIGHT - size || y < 0) {
50             ySpeed = -ySpeed;
51         }
52     }
53
54     /** DrawCanvas (inner class) is a JPanel used for custom drawing */
55     private class DrawCanvas extends JPanel {
56         @Override

```

```

57     public void paintComponent(Graphics g) {
58         super.paintComponent(g); // paint parent's background
59         setBackground(Color.BLACK);
60         g.setColor(Color.BLUE);
61         g.fillOval(x, y, size, size); // draw a circle
62     }
63 }
64
65 /** The entry main method */
66 public static void main(String[] args) {
67     // Run GUI codes in Event-Dispatching thread for thread safety
68     SwingUtilities.invokeLater(new Runnable() {
69         @Override
70         public void run() {
71             new CGBouncingBallSwingTimer(); // Let the constructor do the job
72         }
73     });
74 }
75 }

```

javax.swing.Timer does not provide very accurate timing due to the overhead of event-handling. It probably cannot be used for real-time application such as displaying a clock.

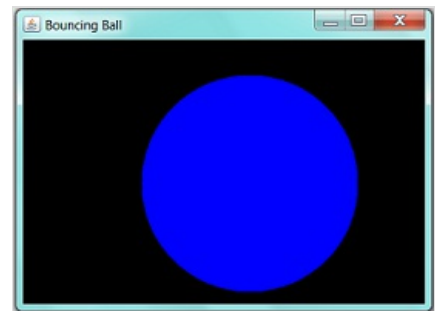
[TODO] Stop the Timer after x steps

8.2 Animation Rules (Advanced)

Animation usually involves multi-threading, so that the GUI refreshing operations does not interfere with the programming logic. Multi-threading is an advanced topics. Read "[Multithreading & Concurrent Programming](#)"

In the previous example, we use javax.swing.Timer, which run the updating task at regular interval on the event-dispatching thread. In this example, we shall create a new thread to run the update.

To create a new thread, define a (anonymous and inner) subclass of Thread and override the run() method to specify the behavior of the task. Create an instance and start the instance via the start() method, which calls back the run() defined earlier.



To ensure the new thread does not starve the other threads, in particular the event-dispatching thread, the thread shall yield control via the sleep(millis) method, which also provides the necessary delay.

```

1  import java.awt.*;
2  import javax.swing.*;
3
4  /** Bouncing Ball (Animation) via custom thread */
5  public class CGBouncingBall extends JFrame {
6      // Define named-constants
7      private static final int CANVAS_WIDTH = 640;
8      private static final int CANVAS_HEIGHT = 480;
9      private static final int UPDATE_INTERVAL = 50; // milliseconds
10
11     private DrawCanvas canvas; // the drawing canvas (extends JPanel)
12
13     // Attributes of moving object
14     private int x = 100; // top-left (x, y)
15     private int y = 100;
16     private int size = 250; // width and height
17     private int xSpeed = 3; // moving speed in x and y directions
18     private int ySpeed = 5; // displacement per step in x and y
19
20     /** Constructor to setup the GUI components */
21     public CGBouncingBall() {
22         canvas = new DrawCanvas();
23         canvas.setPreferredSize(new Dimension(CANVAS_WIDTH, CANVAS_HEIGHT));
24         this.setContentPane(canvas);
25         this.setDefaultCloseOperation(EXIT_ON_CLOSE);
26         this.pack();
27         this.setTitle("Bouncing Ball");
28         this.setVisible(true);
29
30         // Create a new thread to run update at regular interval
31         Thread updateThread = new Thread() {
32             @Override
33             public void run() {
34                 while (true) {
35                     update(); // update the (x, y) position
36                     repaint(); // Refresh the JFrame. Called back paintComponent()

```

```

37         try {
38             // Delay and give other thread a chance to run
39             Thread.sleep(UPDATE_INTERVAL); // milliseconds
40         } catch (InterruptedException ignore) {}
41     }
42 }
43 };
44 updateThread.start(); // called back run()
45 }
46
47 /** Update the (x, y) position of the moving object */
48 public void update() {
49     x += xSpeed;
50     y += ySpeed;
51     if (x > CANVAS_WIDTH - size || x < 0) {
52         xSpeed = -xSpeed;
53     }
54     if (y > CANVAS_HEIGHT - size || y < 0) {
55         ySpeed = -ySpeed;
56     }
57 }
58
59 /** DrawCanvas (inner class) is a JPanel used for custom drawing */
60 class DrawCanvas extends JPanel {
61     @Override
62     public void paintComponent(Graphics g) {
63         super.paintComponent(g); // paint parent's background
64         setBackground(Color.BLACK);
65         g.setColor(Color.BLUE);
66         g.fillOval(x, y, size, size); // draw a circle
67     }
68 }
69
70 /** The entry main method */
71 public static void main(String[] args) {
72     // Run GUI codes in Event-Dispatching thread for thread safety
73     SwingUtilities.invokeLater(new Runnable() {
74         @Override
75         public void run() {
76             new CGBouncingBall(); // Let the constructor do the job
77         }
78     });
79 }
80 }

```

- To update the display regularly, we explicitly invoke the `repaint()` method of the `JFrame`, which will callback the `paintComponent(g)` of all the components contained in this `JFrame`.
- The display refreshing code is run in its own thread, so as to avoid the infamous unresponsive user interface problem. It is programmed as an anonymous inner class, extends class `Thread`, by overriding the `run()` method to provide the programmed operations (i.e., `repaint()`). The `start()` method is use to start the thread, which will callback the `run()`.
- Inside the overridden `run()`, the `repaint()` is programmed inside an infinite loop, followed by a `Thread.sleep(milliseconds)` method, which suspends the thread for the given milliseconds. This operation provides the necessary delay and also yield control to other thread to perform their intended operations.

[TODO] Stopping the thread after x steps

9 . (A d v a n c e d) A r @ p @ s i e n t (L) o k a t

Reference: "Painting in AWT and Swing" @ <http://www.oracle.com/technetwork/java/painting-140037.html>. I summarise some of the important points here.

H e a v y w e i g h t A W T C o m p o n e n t s v s . L i g h t w e i g h t S w i n g

The original AWT components are heavyweight components. "Heavyweight" means that the component has it's own opaque native window. Heavyweight components, such as `java.awt.Button`, are mapped to the platform-specific components. It relies on the windowing subsystem in each native platform to take care of details such as damage detection, clip calculation, and z-ordering. On the other hand, the newer Swing JComponents (such as `javax.swing.JButton`) are lightweight components. A "lightweight" component does not own its screen resources but reuses the native window of its closest heavyweight ancestor. Swing JComponents do not rely on the native platform and are written purely in Java, . The top-level containers, such as `JFrame`, `JApplet` and `JDialog`, which are not subclass of `JComponent`, remain heavyweight. It is because the lightweight Swing JComponents need to attach to a heavyweight ancestor.

P a i n t i n g M e c h a n i s m

Painting is carried out via a "call-back" mechanism. A program shall put its painting codes in a overridden method (`paint()`) for AWT components or

`paintComponent()` for Swing component), and the windowing subsystem will call back this method when it's time to paint.

System-triggered vs. Application-triggered Painting

There are two types of paint (or repaint) requests:

1. System-triggered: e.g., the component is first made visible, the component is resized, etc. The windowing subsystem will schedule `paint()` or `paintComponent()` on the event-dispatching thread.
2. Application-triggered: application has modified the appearance of the component and requested to repaint the component. However, Application shall not invoke `paint()` or `paintComponent()` directly. Instead, it shall invoke a special method called `repaint()`, which will in turn invoke `paint()` or `paintComponent()`. Multiple `repaint()` requests may be collapsed into a single `paint()` call.

Instead of issuing `repaint()` to paint the entire component, for efficiency, you can selectively repaint a rectangular clip area. You can also specify a maximum time limit for painting to take place.

```
public void repaint()
    // requests to repaint this component
public void repaint(long timeMax)
    // repaint before timeMax msec (for lightweight components)
public void repaint(int x, int y, int width, int height)
    // repaint the specified rectangular area
public void repaint(long timeMax, int x, int y, int width, int height)
    // repaint the specified rectangular area within timeMax msec
```

Painting the Lightweight Swing Components

A lightweight needs a heavyweight somewhere up the containment hierarchy in order to have a place to paint, as only heavyweight components have their own opaque window. When this heavyweight ancestor is asked to paint its window, it must also paint all of its lightweight descendents. This is handled by `java.awt.Container`'s `paint()` method, which calls `paint()` on any of its visible, lightweight children which intersect with the rectangle to be painted. Hence, it is crucial for all `Container` subclasses (lightweight or heavyweight) that override `paint()` to place a `super.paint()` call in the `paint()` method. This `super.paint()` call invoke `Container`'s (super) `paint()` method, which in turn invoke `paint()` on all its descendents. If the `super.paint()` call is missing, some of the lightweight descendents will be shown up.

Opaque and Transparent

Lightweight components does not own its opaque window and "borrow" the screen real estate of its heavyweight ancestor. As a result, they could be made transparent, by leaving their background pixels unpainted to allow the underlying component to show through.

To improve performance of opaque components, Swing adds a property called `opaque` to all `JComponents`. If `opaque` is set to `true`, the component agrees to paint all of the pixels contained within its rectangular bounds. In other words, the windowing subsystem does not have to do anything within these bounds such as painting its ancestors. If `opaque` is set to `false`, the component makes no guarantees about painting all the bits within its rectangular bounds, and the windowing subsystem has more work to do.

Swing further factor the `paint()` method into three methods, which are invoked in the following order:

```
protected void paintComponent(Graphics g)
protected void paintBorder(Graphics g)
protected void paintChildren(Graphics g)
```

Swing programs should override `paintComponent()` instead of `paint()`.

Most of the standard Swing components (in particular, `JPanel`) have their look and feel implemented by separate look-and-feel objects (called "UI delegates") for Swing's Pluggable look and feel feature. This means that most or all of the painting for the standard components is delegated to the UI delegate and this occurs in the following way:

1. `paint()` invokes `paintComponent()`.
2. If the `ui` property is non-null, `paintComponent()` invokes `ui.update()`.
3. If the component's `opaque` property is `true`, `ui.update()` fills the component's background with the background color and invokes `ui.paint()`.
4. `ui.paint()` renders the content of the component.

This means that subclasses of Swing components which have a UI delegate (such as `JPanel`), should invoke `super.paintComponent()` within their overridden `paintComponent()`, so that `ui.update()` fills the background (of the superclass such as `JPanel`) provided `opaque` is `true`.

```
public class MyPanel extends JPanel {
    @Override
    protected void paintComponent(Graphics g) {
        // Let UI delegate paint first
        // (including background filling, if I'm opaque)

        super.paintComponent(g); // fill the JPanel's background and invoke ui.paint()

        // paint my contents next...
    }
}
```

```
}
```

Try removing the `super.paintComponent()` from a Swing program that does animation (e.g., bouncing ball). The background will not be painted, and the previous screen may not be cleared. You can also paint the background yourself by filling a `Rectangle` with background color.

```
@Override
protected void paintComponent(Graphics g) {
    g.setColor(backgroundColor);
    g.fillRect(0, 0, getWidth() - 1, getHeight() - 1);
}
```

Furthermore, if you set the opaque to false (via `setOpaque(false)`) for the subclass of `JPanel`, the `super.paintComponent(g)` does not fill the background.

REFERENCES & RESOURCES

- "The Swing Tutorial" @ <http://docs.oracle.com/javase/tutorial/uiswing/>, in particular, the section on "Performing Custom Graphics".
- "Painting in AWT and Swing" @ <http://www.oracle.com/technetwork/java/painting-140037.html>.

Latest version tested: JDK 1.7.0_17
Last modified: April, 2013

Feedback, comments, corrections, and errata can be sent to Chua Hock-Chuan (ehchua@ntu.edu.sg) | [HOME](#)