TABLE OF CONTENTS (HIDE)

1. Revisit `java.awt.Graphics` f
1.1 Template for Custom Drawing
2. Java 2D API & `Graphics2D`
2.1 `java.awt.Graphics2D`
2.2 Affine Transform (`java.awt.`
2.3 Geometric Primitives and Shap
2.4 `Point2D` and its Subclasses P
2.5 Interface `java.awt.Shape`
2.6 `Stroke`, `Paint` and `Composi`
3. Working with Bitmap Images
3.1 Loading Images
3.2 `drawImage()`
3.3 Image Affine Transforms
3.4 Image Filtering Operations
3.5 Animating Image Frames
4. High Performance Graphics
4.1 Full-Screen Display Mode (JDK
4.2 Rendering to the Display & Dc
4.3 Splash Screen

Java Game Programming

2D Graphics, Java2D and Images

The pre-requisite knowledge needed to for game programming includes:

- OOP, in particular the concepts of inheritance and polymorphism for designing classes.
- GUI and custom graphics programming (`javax.swing`).
- Event-handling, in particular mouse-event and key-event handling (`java.awt.event`).
- Graphics programming using Java 2D (`java.awt.geom`).
- Paying sounds (`javax.sound`).
- Basic knowledge on I/O, multi-threading for starting the game thread, and timing control.
- Transformation, collision detection and reaction algorithms.

Advanced Knowledge:

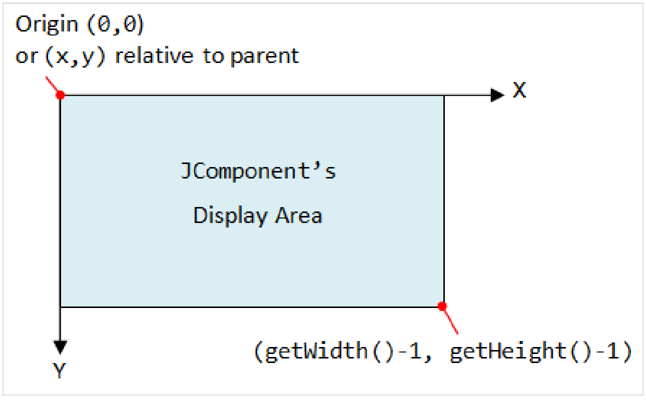
- JOGL (Java Bindings to OpenGL), or Java3D for 3D graphics.
- JOAL (Java Bindings to OpenAL) for advanced sound.

1. Revisit `java.awt.Graphics` for Custom Drawing

Read: "Custom Graphics" chapter.

The `java.awt.Graphics` class is used for custom painting. It manages the graphics context (such as color, font and clip area) and provides methods for rendering of three types of graphical objects:

1. Texts: via `drawString()`.
2. Vector-graphic primitives and shapes: via `drawXxx()` and `fillXxx()` for `Line`, `PolyLine`, `Oval`, `Rect`, `RoundRect`, `3DRect`, and `Arc`.
3. Bitmap images: via `drawImage()`.



The `Graphics` class also allows you to get/set the attributes of the graphics context:

- Font (`setFont()`, `getFont()`)
- Color (`setColor()`, `getColor()`)
- Display clipping area (`getClip()`, `getClipBounds()`, `setClip()`)

It is important to take note that the graphics co-ordinates system is "inverted", as illustrated.

The `java.awt.Graphics` class is, however, limited in its functions and capabilities. It supports mainly straight-line segments. `java.awt.Graphics2D` (of the Java 2D API) greatly extends the functions and capabilities of the `Graphics` class.

1.1 Template for Custom Drawing

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

// Swing Program Template
@SuppressWarnings("serial")
public class SwingTemplateJPanel extends JPanel {
   // Name-constants
   public static final int CANVAS_WIDTH = 640;
   public static final int CANVAS_HEIGHT = 480;
   public static final String TITLE = "...Title...";
   // ......

   // private variables of GUI components
   // ......

   /** Constructor to setup the GUI components */
   public SwingTemplateJPanel() {
      setPreferredSize(new Dimension(CANVAS_WIDTH, CANVAS_HEIGHT));
      // "this" JPanel container sets layout
      // setLayout(new ....Layout());

      // Allocate the UI components
      // .....

      // "this" JPanel adds components
      // add(....)

      // Source object adds listener
      // .....
   }

   /** Custom painting codes on this JPanel */
   @Override
   public void paintComponent(Graphics g) {
      super.paintComponent(g);  // paint background
      setBackground(Color.BLACK);

      // Your custom painting codes
      // ......
   }

   /** The entry main() method */
   public static void main(String[] args) {
      // Run GUI codes in the Event-Dispatching thread for thread safety
      SwingUtilities.invokeLater(new Runnable() {
         public void run() {
            JFrame frame = new JFrame(TITLE);
            frame.setContentPane(new SwingTemplateJPanel());
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.pack();             // "this" JFrame packs its components
            frame.setLocationRelativeTo(null); // center the application window
            frame.setVisible(true);            // show it
         }
      });
   }
}
```

Dissecting the Program

- The custom drawing is done by extending a `JPanel` and overrides the `paintComponent()` method.
- The size of the drawing panel is set via the `setPreferredSize()`.
- The `JFrame` does not set its size, but packs its components by invoking `pack()`.
- In the `main()`, the GUI construction is carried out in the event-dispatch thread to ensure thread-safety.

You can run the above class as an applet, by providing a main applet class:

```java
import javax.swing.*;

/** Swing Program Template for running as Applet */
@SuppressWarnings("serial")
public class SwingTemplateApplet extends JApplet {

   /** init() to setup the UI components */
   @Override
   public void init() {
      // Run GUI codes in the Event-Dispatching thread for thread safety
      try {
         SwingUtilities.invokeAndWait(new Runnable() { // Applet uses invokeAndWait()
            @Override
            public void run() {
               // Set the content-pane of the JApplet to an instance of main JPanel
               setContentPane(new SwingTemplateJPanel());
            }
         });
      } catch (Exception e) {
         e.printStackTrace();
      }
   }
}
```

2. Java 2D API & `Graphics2D`

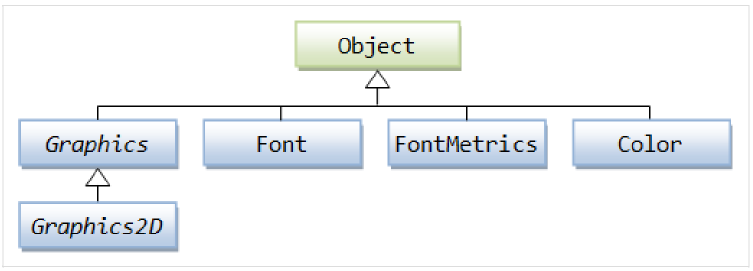
Reference: Java Tutorial's "2D Graphics" @ http://docs.oracle.com/javase/tutorial/2d/TOC.html.

Java AWT API has been around since JDK 1.1. Java 2D API is part of Java Foundation Class (JFC), similar to Swing, and was introduced in JDK 1.2. It provides more capabilities for graphics programming. Java 2D spans many packages: `java.awt`, `java.awt.image`, `java.awt.color`, `java.awt.font`, `java.awt.geom`, `java.awt.print`, and `java.awt.image.renderable`.

2.1 `java.awt.Graphics2D`

The core class in Java2D is the `java.awt.Graphics2D`. `Graphics2D` is a subclass of `java.awt.Graphics`, which extends the support of the legacy `Graphics` class in rendering three groups of objects: text, vector-graphics and bitmap images. It also supports more attributes that affect the rendering, e.g.,

- Transform attribute (translation, rotation, scaling and shearing).
- Pen attribute (outline of a shape) and Stroke attribute (point-size, dashing-pattern, end-cap and join decorations).
- Fill attribute (interior of a shape) and Paint attribute (fill shapes with solid colors, gradients, and patterns).
- Composite attribute (for overlapping shapes).
- Clip attribute (display area).
- Font attribute.



`Graphics2D` is designed as a subclass of `Graphics`. To use `Graphcs2D` context, *downcast* the `Graphics` handle `g` to `Graphics2D` in painting methods such as `paintComponent()`. This works because the graphics subsystem in fact passes a `Graphics2D` object as the argument when calling back the painting methods. For example,

```
@Override
public void paintComponent(Graphics g) {  // graphics subsystem passes a Graphis2D subclass object as argument
   super.paintComponent(g);            // paint parent's background
   Graphics2D g2d = (Graphics2D) g;    // downcast the Graphics object back to Graphics2D
   // Perform custom drawing using g2d handle
   ......
}
```

Besides the `drawXxx()` and `fillXxx()`, `Graphics2D` provides more general drawing methods which operates on `Shape` interface.

```
public abstract void draw(Shape s)
public abstract void fill(Shape s)
public abstract void clip(Shape s)
```

2.2 Affine Transform (java.awt.geom.AffineTransform)

Transform is key in game programming and animation!

An *affine transform* is a linear transform such as *translation*, *rotation*, *scaling*, or *shearing* in which a straight line remains straight and parallel lines remain parallel after the transformation. It can be represented using the following 3x3 matrix operation:

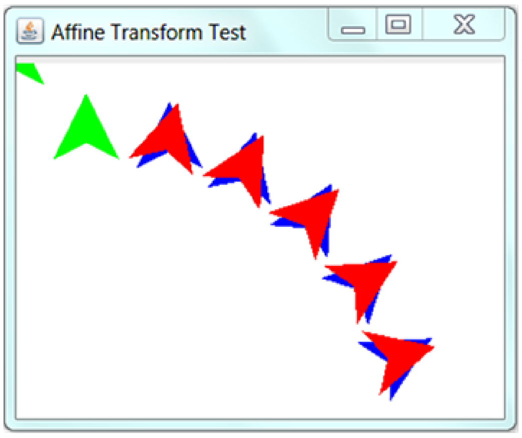
```
[ x' ]   [ m00  m01  m02 ] [ x ]   [ m00x + m01y + m02 ]
[ y' ] = [ m10  m11  m12 ] [ y ] = [ m10x + m11y + m12 ]
[ 1  ]   [   0    0    1 ] [ 1 ]   [         1         ]
```

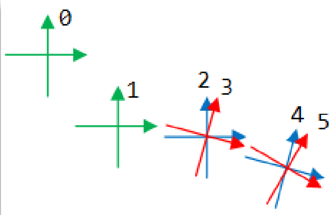
Affine transform is supported via the `java.awt.geom.AffineTransform` class. The `Graphics2D` context maintains an `AffineTransform` attribute, and provides methods to change the transform attributes:

```
// in class java.awt.Graphics2D
public abstract void setTransform(AffineTransform at);   // overwrites the current Transform in the Graphics2D context
public abstract void translate(double tx, double ty);    // concatenates the current Transform with a translation transform
public abstract void rotate(double theta);               // concatenates the current Transform with a rotation transform
public abstract void scale(double scaleX, double scaleY) // concatenates the current Transform with a scaling transform
public abstract void shear(double shearX, double shearY) // concatenates the current Transform with a shearing transform
```

Example

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

/** Test applying affine transform on vector graphics */
@SuppressWarnings("serial")
public class AffineTransformDemo extends JPanel {
   // Named-constants for dimensions
   public static final int CANVAS_WIDTH = 640;
   public static final int CANVAS_HEIGHT = 480;
   public static final String TITLE = "Affine Transform Demo";

   // Define an arrow shape using a polygon centered at (0, 0)
   int[] polygonXs = { -20, 0, +20, 0};
   int[] polygonYs = { 20, 10, 20, -20};
   Shape shape = new Polygon(polygonXs, polygonYs, polygonXs.length);
   double x = 50.0, y = 50.0;  // (x, y) position of this Shape

   /** Constructor to set up the GUI components */
   public AffineTransformDemo() {
      setPreferredSize(new Dimension(CANVAS_WIDTH, CANVAS_HEIGHT));
   }

   /** Custom painting codes on this JPanel */
   @Override
   public void paintComponent(Graphics g) {
      super.paintComponent(g);    // paint background
      setBackground(Color.WHITE);
      Graphics2D g2d = (Graphics2D)g;

      // Save the current transform of the graphics contexts.
      AffineTransform saveTransform = g2d.getTransform();
      // Create a identity affine transform, and apply to the Graphics2D context
      AffineTransform identity = new AffineTransform();
      g2d.setTransform(identity);

      // Paint Shape (with identity transform), centered at (0, 0) as defined.
      g2d.setColor(Color.GREEN);
      g2d.fill(shape);
      // Translate to the initial (x, y) position, scale, and paint
      g2d.translate(x, y);
      g2d.scale(1.2, 1.2);
      g2d.fill(shape);

      // Try more transforms
      for (int i = 0; i < 5; ++i) {
         g2d.translate(50.0, 5.0);  // translates by (50, 5)
         g2d.setColor(Color.BLUE);
         g2d.fill(shape);
         g2d.rotate(Math.toRadians(15.0)); // rotates about transformed origin
         g2d.setColor(Color.RED);
         g2d.fill(shape);
      }
      // Restore original transform before returning
      g2d.setTransform(saveTransform);
   }

   /** The Entry main method */
   public static void main(String[] args) {
      // Run the GUI codes on the Event-Dispatching thread for thread safety
      SwingUtilities.invokeLater(new Runnable() {
         @Override
         public void run() {
            JFrame frame = new JFrame(TITLE);
            frame.setContentPane(new AffineTransformDemo());
            frame.pack();
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.setLocationRelativeTo(null); // center the application window
            frame.setVisible(true);
         }
      });
   }
}
```





Co-ordinates after each transform

These are the steps for carrying out affine transform with a `Graphics2D` context:

1. Override the `paintComponent(Graphics)` method of the custom drawing `JPanel`. Downcast the `Graphics` object into `Graphics2D`.

```
@Override
public void paintComponent(Graphics g) {
   Graphics2D g2d = (Graphics2D) g;
   ......
}
```

2. Prepare a `Shape` (such as `Polygon`, `Rectangle2D`, `Rectangle`). The original shape shall center at (0, 0) to simplify rotation.
3. Save the current transform associated with this `Graphics2D` context, and restore the saved transform before exiting the method.

```
AffineTransform saveTransform = g2d.getTransform();    // save
....
g2d.setTransform(saveTransform);                       // restore
```

4. Allocate a new `AffineTransform`. Initialize to the default *identity* transform. Apply it to the current `Graphics2D` context. You can then use the `Graphics2D` context to perform translation, rotation, scaling and shearing.

```
AffineTransform identity = new AffineTransform();  // an identity transform
g2d.setTransform(identity);  // overwrites the transform associated with this Graphics2D context
g2d.translate(x, y);         // translates from (0, 0) to the current (x, y) position
g2d.scale(scaleX, scaleY);   // scaling
g2d.rotate(angle);           // rotation clockwise about (0, 0), by angle (in radians)
g2d.shear(shearX, shearY);   // shearing
```

Take note that successive transforms are *concatenated*, until it is reset (to the identity transform) or overwritten.

In the above example, the `Polygon`, which is originally centered at (0, 0) (shown in green), is first translated to (50, 50) and scaled up by 1.2 (in green). A loop of 5-iterations is applied to translate by (50, 5) (in blue) and rotate by 15 degrees about (0, 0) (in red). Observe that after each transform, the *axes* and *origin* are transformed accordingly. This is especially noticeable for rotation, as the axes are no longer parallel and perpendicular to the screen and its origin is shifted as well. The origin of the axes is set to (0, 0) by the identity transform.

Rotation

The result of rotation depends on the angle rotated, as well as its rotation center. Two methods are provided:

```
public abstract void rotate(theta);
public abstract void rotate(theta, anchorX, anchorY)
```

The first method rotates about the origin (0, 0), while the second method rotate about (anchorX, anchorY), without affecting the origin after the rotation. Take note that after each rotation, the coordinates is rotated as well. Rotation can be made simpler, by center the shape at (0, 0) as shown in the above example.

2.3 Geometric Primitives and Shapes

Java 2D's primitives include:

- point (`Point2D`)
- line (`Line2D`)
- rectangular shapes (`Rectangle2D`, `RoundRectangle2D`,, `Ellipse2D`, `Arc2D`, `Dimension2D`)
- quadratic and cubic curves with control points (`QuadCurve2D`, `CubicCurve2D`)
- arbitrary shapes (`Path2D`).

[TODO] Class diagram

The `Xxx2D` classes have two nested `public static` subclasses `Xxx2D.Double` and `Xxx2D.Float` to support `double`- and `float`-precision. High-quality 2D rendering (e.g., rotation, shearing, curve segments) cannot be preformed in `int` (even though eventually it will be converted to integral screen-pixel values for display). Hence, `double` and `float` subclasses are introduced in Java 2D for better accuracy and smoothness. The earlier `int`-precision classes, such as `Point` and `Rectangle` are retrofitted as a subclass of `Point2D` and `Rectangle2D`.

The `Xxx2D`, `Xxx2D.Double` and `Xxx2D.Float` are kept in package `java.awt.geom` package.

GeneralPath

The `java.awt.geom.GeneralPath` class represents a geometric path constructed from straight lines, quadratic and cubic curves. For example,

```
int[] x = { -20, 0, 20, 0};
int[] y = { 20, 10, 20, -20};
GeneralPath p = new GeneralPath();
p.moveTo(x[0], y[0]);
for (int i = 1; i < x.length; ++i) {
   p.lineTo(x[i], y[i]);
}
p.closePath();
g2d.translate(350, 250);
g2d.draw(p);
```

2.4 `Point2D` and its Subclasses `Point`, `Point2D.Double` and `Point2D.Float` (Advanced)

Reference: Source codes of `java.awt.geom.Point2D` and `java.awt.Point`.

As an example, let's examine `Point2D abstract` superclass and its subclasses more closely.

java.awt.geom.Point2D

`Point2D` is an abstract class that cannot be instantiated. It declares the following `abstract` methods:

```
abstract public double getX();
abstract public double getY();
abstract public void setLocation(double x, double y);
```

It also defines and implemented some `static` methods and member methods. (Hence, it is designed as an abstract class, instead of interface.)

```
// static methods
public static double distance(double x1, double y1, double x2, double y2)
public static double distanceSq(double x1, double y1, double x2, double y2)
// instance (non-static) methods
public double distance(double x, double y)
public double distanceSq(double x, double y)
public double distance(Point2D p)
public double distanceSq(Point2D p)
......
```

`Point2D` does not define any instance variable, in particular, the `x` and `y` location of the point. This is because it is not sure about the type of `x` and `y` (which could be `int`, `float` or `double`). The instance variables, therefore, are left to the implementation subclasses. Three subclasses were implemented for types of `int`, `float` and `double`, respectively.

Subclasses `java.awt.Point`, `java.awt.geom.Point2D.Double` and `java.awt.geom.Point2D.Float`

How to design these subclasses?

java.awt.Point is the subclass for int-precision. It declares instance variables x and y as int, provides implementation to the abstract methods declared in the superclass, and provides some additional methods.

```
package java.awt;
public class Point extends Point2D {
   // Instance variables (x, y) of type int
   public int x;
   public int y;
   // Constructor
   public Point(int x, int y) { this.x = x; this.y = y; }
   // Provide implementation to the abstract methods declared in the superclass
   public double getX() { return x; }
   public double getY() { return y; }
   public void setLocation(double x, double y) {
      this.x = (int)Math.floor(x + 0.5);
      this.y = (int)Math.floor(y + 0.5);
   }
   // Other methods
   ......
}
```

Point (of int-precision) is a straight-forward implementation of inheritance and polymorphism. Point is a legacy class (since JDK 1.1) and retrofitted when Java 2D was introduced. For higher-quality and smoother graphics (e.g., rotation), int-precision is insufficient. Java 2D, hence, introduced float-precision and double-precision points.

Point2D.Double (for double-precision point) and Point2D.Float (for float-precision point) are, however, implemented as nested public static subclasses inside the Point2D outer class.

Recall that nested class (or inner class) can be used for:

1. Simplifying access control: inner class can access the private variables of the enclosing outer class, as they are at the same level.
2. Keeping codes together and namespace management.

In this case, it is used for namespace management, as there is no access-control involved.

```
package java.awt.geom;
abstract public class Point2D {  // outer class

   public static class Double extend Point2D {  // public static nested class and subclass
      public double x;
      public double y;
      // Constructor
      public Double(double x, double y) { this.x = x; this.y = y; }
      // Provides implementation to the abstract methods
      public double getX() { return x; }
      public double getY() { return y; }
      public void setLocation(double x, double y) {
         this.x = x;
         this.y = y;
      }
      // Other methods
      ......
   }

   public static class Float extend Point2D {  // public static nested class and subclass
      public float x;
      public float y;
      // Constructor
      public Float(float x, float y) { this.x = x; this.y = y; }
      // Provide implementation to the abstract methods
      public double getX() { return x; }
      public double getY() { return y; }
      public void setLocation(double x, double y) {
         this.x = (float)x;
         this.y = (float)y;
      }
      // Other methods
      ......
   }

   // Definition for the outer class
   abstract public double getX();
   abstract public double getY();
   abstract public void setLocation(double x, double y);
   ......
}
```

Double and Float are static. In other words, they can be referenced via the classname as Point2D.Double and Point2D.Float, and used directly without instantiating the outer class, just like any static variable or method (e.g., Math.PI, Math.sqrt(), Integer.parseInt()). They are subclasses of Point2D, and thus can be upcasted to Point2D.

```
Point2D.Double p1 = new Point2D.Double(1.1, 2.2);
Point2D.Float  p2 = new Point2D.Float(3.3f, 4.4f);
Point          p3 = new Point(5, 6);
// You can upcast subclasses Point2D.Double, Point2D.Float and Point to superclass Point2D.
```

```
Point2D p4 = new Point2D.Double(1.1, 2.2);
Point2D p5 = new Point2D.Float(3.3f, 4.4f);
Point2D p6 = new Point(5, 6);
```

In summary, you can treat `Point2D.Double` and `Point2D.Float` as *ordinary* classes with a slightly longer name. They were designed as nested class for namespace management, instead of calling them `Point2DDouble` and `Point2DFloat`.

Note: These classes were designed before JDK 1.5, which introduces generic to support passing of type.

2.5 Interface `java.awt.Shape`

Almost all the `Xxx2D` classes (except `Point2D`) implements `java.awt.Shape` interface. It can be used as argument in `Graphics2D`'s `draw(`*`Shape`*`)` and `fill(`*`Shape`*`)` methods.

The `Shape` interface declares `abstract` methods `contains()` and `intersects()`, which are useful in game programming for collision detection:

```
// Is this point within this Shape's bounds?
boolean contains(double x, double y)
boolean contains(Point2D p)
// Is this rectangle within this Shape's bounds?
boolean contains(double x, double y, double width, double height)
boolean contains(Rectangle2D rect)
// Does the interior of the given bounding rectangle intersect with this shape?
boolean intersects(double x, double y, double width, double height)
boolean intersects(Rectangle2D rect)
```

The `Shape` interface also declares methods to return the bounding rectangle of this shape (again, useful in collision detection).

```
Rectangle getBounds()       // int-precision
Rectangle2D getBounds2D()   // higher precision version
```

The `Shape` interface declares a method called `getPathIterator()` to retrieve a `PathIterator` object that can be used to iterate along the `Shape` boundary.

```
PathIterator getPathIterator(AffineTransform at)
PathIterator getPathIterator(AffineTransform at, double flatness)
```

The popular legacy `java.awt.Polygon` class was retrofitted to implement `Shape` interface. However, there is no `Polyon2D` in Java 2D, which can be served by a more generic `Path2D`.

[TODO] Example

2.6 `Stroke`, `Paint` and `Composite` Attributes

Pen's Stroke

The `Graphisc2D`'s stroke attribute control the pen-stroke used for the outline of a shape. It is set via the `Graphis2D`'s `setStroke()`. A `Stroke` object implements `java.awt.Stroke` interface. Java 2D provides a built-in `java.awt.BasicStroke`.

```
public BasicStroke(float width, int cap, int join, float miterlimit, float[] dash, float dash_phase)
   // All parameters are optional
   // width:  width of the pen stroke
   // cap: the decoration of the ends, CAP_ROUND, CAP_SQUARE or CAP_BUTT.
   // join: the decoration where two segments meet, JOIN_ROUND, JOIN_MITER, or JOIN_BEVEL
   // miterlimit: the limit to trim the miter join.
   // dash: the array representing the dashing pattern.
   // dash_phase: the offset to start the dashing pattern
```

For example,

```
g2d.setStroke(new BasicStroke(10, BasicStroke.CAP_ROUND, BasicStroke.JOIN_ROUND));
g2d.setColor(Color.CYAN);
g2d.draw(new Rectangle2D.Double(300, 50, 200, 100));

// Test dash-stroke
float[] dashPattern = {20, 5, 10, 5};  // dash, space, dash, space
g2d.setStroke(new BasicStroke(5, BasicStroke.CAP_BUTT, BasicStroke.JOIN_ROUND,
      10, dashPattern, 0));
g2d.setColor(Color.CYAN);
g2d.draw(new Rectangle2D.Double(50, 200, 200, 100));
```

Paint

The `Graphics2D`'s paint attribute determines the color used to render the shape. It is set via the `Graphics2D`'s `setPaint()` method. A `Paint` object implements the `java.awt.Paint` interface. Java 2D provides many built-in `Paint` objects such as `GradientPaint`, `LinearGradientPaint`, `RadialGradientPaint`, `MultipleGradientPaint`, `TexturePaint`, and others.

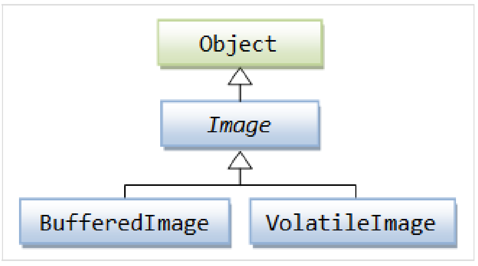
For example,

```
g2d.setPaint(new GradientPaint(50, 80, Color.RED, 250, 180, Color.GREEN));
   // set current paint context to a GradientPaint, from (x1, y1) with color1 to (x2, y2) with color2
g2d.fill(new Rectangle2D.Double(50, 80, 200, 100));
   // fill the Shap with the current paint context
```

Composite

[TODO] How to compose the drawing of primitive with the underlying graphics area.

3. Working with Bitmap Images



Reference: Java Tutorial's "2D Graphics" Section "Working with Images" @ http://docs.oracle.com/javase/tutorial/2d/images/index.html"

A bitmap image is a 2D rectangular array of *pixels*. Each pixel has a *color* value (typically in RGB or RGBA). The dimension of the image is represented by its *width* and *length* in pixels. In Java, the origin (0, 0) of an image is positioned at the top-left corner, like all other components.

Most of the image display and processing methods work on `java.awt.Image`. `Image` is an `abstract` class that represent an image as a rectangular array of pixels. The most commonly-used implementation subclass is `java.awt.image.BufferedImage`, which stores the pixels in memory buffer so that they can be directly accessed. A `BufferedImage` comprises a `ColorModel` and a `Raster` of pixels. The `ColorModel` provides a color interpretation of the image's pixel data.

An `Image` object (and subclass `BufferedImage`) can be rendered onto a `JComponent` (such as `JPanel`) via `Graphics`' (or `Graphics2D`') `drawImage()` method.

Image is typically read in from an external image file into a `BufferedImage` (although you can create a `BufferedImage` based on algorithm). Image file formats supported by JDK include:

- GIF
- PNG (Portable Network Graphics)
- JPEG
- BMP

`BufferedImage` supports image filtering operations (such as convolution). The resultant image can be painted on the screen, sent to printer, or save to an external file.

Transparent vs. Opaque Background

PNG and GIF supports transparent background. JPEG does not. PNG and GIF are palette-based. They maintain a list of palettes and map each palette number to a RGB color value. Every image pixel is then labeled with a palette number, which is then mapped to the actual RGB values. One of the palette number can be designated as *transparent*. Pixels with this transparent palette value will not be displayed. Instead, the background of the image is displayed.

3.1 Loading Images

There are several ways to create an `Image` for use in your program.

Using `ImageIO.read()` (JDK 1.4)

The easiest way to load an image into your program is to use the `static` method `ImageIO.read()` of the `javax.imageio.ImageIO` class, which returns a `java.awt.image.BufferedImage`. Similar, you can use `ImageIO.write()` to write an image.

```
public static BufferedImage read(URL imagePath) throws IOException
public static BufferedImage read(File imagePath) throws IOException
public static BufferedImage read(InputStream stream) throws IOException
public static BufferedImage read(ImageInputStream stream) throws IOException
```

Instead of using `java.io.File` class to handle a disk file, it is better to use `java.net.URL`. `URL` is more flexible and can handle files from more sources, such as disk file and JAR file (used for distributing your program). It works on application as well as applet. For example,

```
import java.awt.*;
import java.io.IOException;
import java.net.URL;
import javax.imageio.ImageIO;
import javax.swing.*;

/** Test loading an external image into a BufferedImage using ImageIO.read() */
@SuppressWarnings("serial")
public class LoadImageDemo extends JPanel {
   // Named-constants
   public static final int CANVAS_WIDTH = 640;
   public static final int CANVAS_HEIGHT = 480;
   public static final String TITLE = "Load Image Demo";

   private String imgFileName = "images/duke.gif"; // relative to project root (or bin)
   private Image img;  // a BufferedImage object

   /** Constructor to set up the GUI components */
   public LoadImageDemo() {
      // Load an external image via URL
      URL imgUrl = getClass().getClassLoader().getResource(imgFileName);
      if (imgUrl == null) {
         System.err.println("Couldn't find file: " + imgFileName);
      } else {
         try {
            img = ImageIO.read(imgUrl);
         } catch (IOException ex) {
            ex.printStackTrace();
         }
      }

      setPreferredSize(new Dimension(CANVAS_WIDTH, CANVAS_HEIGHT));
   }

   /** Custom painting codes on this JPanel */
   @Override
   public void paintComponent(Graphics g) {
      super.paintComponent(g);    // paint background
      setBackground(Color.WHITE);
      g.drawImage(img, 50, 50, null);
   }

   /** The Entry main method */
   public static void main(String[] args) {
      // Run the GUI codes on the Event-Dispatching thread for thread safety
      SwingUtilities.invokeLater(new Runnable() {
         @Override
         public void run() {
            JFrame frame = new JFrame("Load Image Demo");
            frame.setContentPane(new LoadImageDemo());
            frame.pack();
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.setVisible(true);
         }
      });
   }
}
```

Using `Toolkit`'s `getImage()`

```
// In java.awt.Toolkit
public abstract Image getImage(URL url)
public abstract Image getImage(String filename)
```

For example,

```
import java.awt.Toolkit;
......
Toolkit tk = Toolkit.getDefaultToolkit();
Image img = tk.getImage("images/duke.gif");
```

Via `ImageIcon`'s `getImage()`

`ImageIcon` is used to decorate `JComponent`s (such as `JLabel` and `JButton`). Construct an `ImageIcon` and get an `Image` via `ImageIcon`'s `getImage()`. For example,

```
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);
}
Image img = icon.getImage();
```

On the other hand, you can also construct an `ImageIcon` from an `Image` object via construtor:

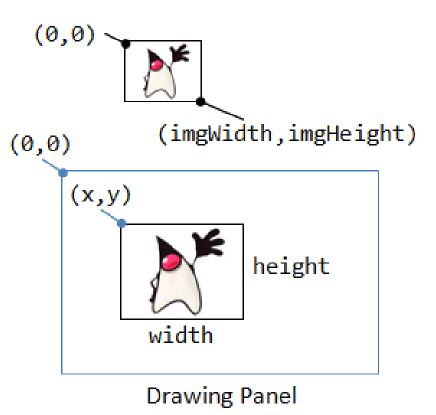
```
public ImageIcon(Image image)   // Construct an ImageIcon from the Image object
```

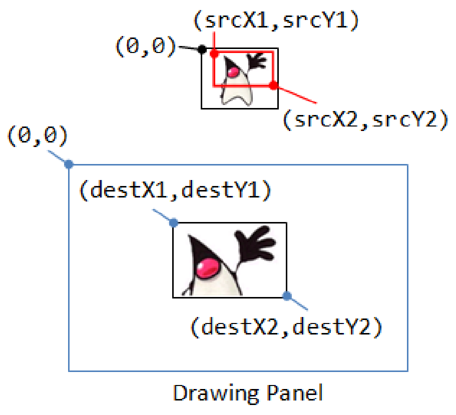
[TODO] Benchmark these methods for small and large images.

3.2 `drawImage()`

The `java.awt.Graphics` class declares 6 versions of `drawImage()` for drawing bitmap images. The subclass `java.awt.Graphics2D` adds a few more.

```
// In class java.awt.Graphics
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 (scrX1, scrY2) and (scrX2, srcY2) is scaled and drawn from
   // (destX1, destY1) to (destX2, destY2).
```





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 the custom drawing `JPanel` (via `this`). The `ImageObserver` is not needed for `BufferedImage`, and shall be set to `null`.

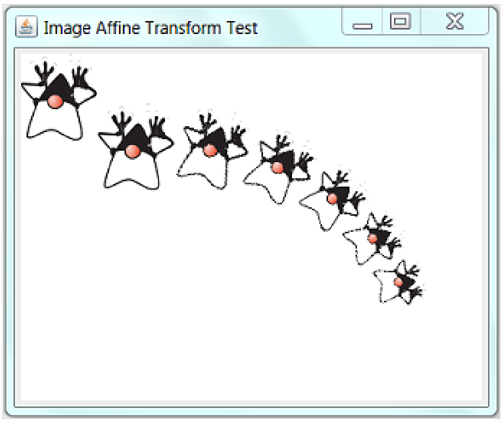
Graphics2D's drawImage()

`Graphics2D` supports affine transfrom and image filtering operations on images, as follows:

```
// In class java.awt.Graphics2D
public abstract boolean drawImage(Image img, AffineTransform xform, ImageObserver obs)
   // Apply the specified AffineTransform to the image
public abstract void drawImage(BufferedImage img, BufferedImageOp op, int x, int y)
   // Apply the specified image filtering operation to the image

public abstract void drawRenderedImage(RenderedImage img, AffineTransform xform)
public abstract void drawRenderableImage(RenderableImage img, AffineTransform xform)
```

3.3 Image Affine Transforms

Java 2D's affine transform works on bitmap image as well as vector graphics. However, instead of manipulating the `Graphics2D`'s current transform context (which operates on vector-graphics only via rendering methods `drawXxx()` and `fillXxx()`), you need to allocate an `AffineTransform` object to perform transformation on images.



Code Example

```
import java.awt.geom.AffineTransform;
import javax.imageio.ImageIO;
import java.net.URL;
import java.awt.*;
import javax.swing.*;
import java.io.*;

/** Test applying affine transform on images */
@SuppressWarnings("serial")
public class ImageTransformDemo extends JPanel {
   // Named-constants for dimensions
   public static final int CANVAS_WIDTH = 640;
   public static final int CANVAS_HEIGHT = 480;
   public static final String TITLE = "Image Transform Demo";

   // Image
   private String imgFileName = "images/duke.png"; // relative to project root or bin
   private Image img;
   private int imgWidth, imgHeight;    // width and height of the image
   private double x = 100.0, y = 80.0; // center (x, y), with initial value

   /** Constructor to set up the GUI components */
   public ImageTransformDemo() {
      // URL can read from disk file and JAR file
      URL url = getClass().getClassLoader().getResource(imgFileName);
      if (url == null) {
         System.err.println("Couldn't find file: " + imgFileName);
      } else {
         try {
            img = ImageIO.read(url);
            imgWidth = img.getWidth(this);
            imgHeight = img.getHeight(this);
         } catch (IOException ex) {
            ex.printStackTrace();
         }
      }

      this.setPreferredSize(new Dimension(CANVAS_WIDTH, CANVAS_HEIGHT));
   }

   /** Custom painting codes on this JPanel */
   @Override
   public void paintComponent(Graphics g) {
      super.paintComponent(g);    // paint background
      setBackground(Color.WHITE);

      Graphics2D g2d = (Graphics2D) g;
      g2d.drawImage(img, 0, 0, this);  // Display with top-left corner at (0, 0)

      // drawImage() does not use the current transform of the Graphics2D context
      // Need to create a AffineTransform and pass into drawImage()
      AffineTransform transform = new AffineTransform();  // identity transform
      // Display the image with its center at the initial (x, y)
      transform.translate(x - imgWidth/2, y - imgHeight/2);
      g2d.drawImage(img, transform, this);
      // Try applying more transform to this image
      for (int i = 0; i < 5; ++i) {
         transform.translate(70.0, 5.0);
         transform.rotate(Math.toRadians(15), imgWidth/2, imgHeight/2); // about its center
         transform.scale(0.9, 0.9);
         g2d.drawImage(img, transform, this);
      }
   }

   /** The Entry main method */
   public static void main(String[] args) {
      // Run the GUI codes on the Event-Dispatching thread for thread safety
      SwingUtilities.invokeLater(new Runnable() {
         @Override
         public void run() {
            JFrame frame = new JFrame(TITLE);
            frame.setContentPane(new ImageTransformDemo());
            frame.pack();
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.setLocationRelativeTo(null); // center the application window
            frame.setVisible(true);
         }
      });
   }
}
```

3.4 Image Filtering Operations

`Graphics2D` supports image filtering operations via the following `drawImage()` method:

```
public abstract void drawImage(BufferedImage img, BufferedImageOp op, int x, int y)
   // Apply the specified image filtering operation to the image
```

Many built-in image filtering operations are available in `java.awt.image` package.

[TODO] more and example

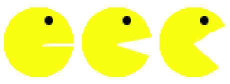
3.5 Animating Image Frames

There are two ways to organize animated image frames:

1. Keep each of the frames in its own file.
2. Keep all of the frames in a stripe (1D or 2D) in a single file for better organization and faster loading.

Code Example 1: Each Frame in its Own File

Three image frames (in its own file) was used in this example, as follow:



In a typical game, the actor has a `(x, y)` position, move at a certain `speed` (in pixels per move-step) and `direction` (in degrees), and may rotate at a `rotationSpeed` (in degrees per move-step).

```
import java.awt.geom.AffineTransform;
import javax.imageio.ImageIO;
import java.net.URL;
import java.awt.*;
import javax.swing.*;
import java.io.*;

```

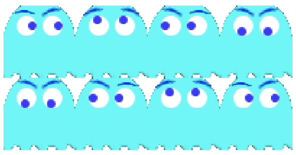
```java
/** Animating image frames. Each frame has its own file */
@SuppressWarnings("serial")
public class AnimatedFramesDemo extends JPanel {
   // Named-constants
   static final int CANVAS_WIDTH = 640;
   static final int CANVAS_HEIGHT = 480;
   public static final String TITLE = "Animated Frame Demo";

   private String[] imgFilenames = {
         "images/pacman_1.png", "images/pacman_2.png", "images/pacman_3.png"};
   private Image[] imgFrames;    // array of Images to be animated
   private int currentFrame = 0; // current frame number
   private int frameRate = 5;    // frame rate in frames per second
   private int imgWidth, imgHeight;    // width and height of the image
   private double x = 100.0, y = 80.0; // (x,y) of the center of image
   private double speed = 8;           // displacement in pixels per move
   private double direction = 0;       // in degrees
   private double rotationSpeed = 1;   // in degrees per move

   // Used to carry out the affine transform on images
   private AffineTransform transform = new AffineTransform();

   /** Constructor to set up the GUI components */
   public AnimatedFramesDemo() {
      // Setup animation
      loadImages(imgFilenames);
      Thread animationThread = new Thread () {
         @Override
         public void run() {
            while (true) {
               update();   // update the position and image
               repaint();  // Refresh the display
               try {
                  Thread.sleep(1000 / frameRate); // delay and yield to other threads
               } catch (InterruptedException ex) { }
            }
         }
      };
      animationThread.start();  // start the thread to run animation

      // Setup GUI
      setPreferredSize(new Dimension(CANVAS_WIDTH, CANVAS_HEIGHT));
   }

   /** Helper method to load all image frames, with the same height and width */
   private void loadImages(String[] imgFileNames) {
      int numFrames = imgFileNames.length;
      imgFrames = new Image[numFrames];  // allocate the array
      URL imgUrl = null;
      for (int i = 0; i < numFrames; ++i) {
         imgUrl = getClass().getClassLoader().getResource(imgFileNames[i]);
         if (imgUrl == null) {
            System.err.println("Couldn't find file: " + imgFileNames[i]);
         } else {
            try {
               imgFrames[i] = ImageIO.read(imgUrl);  // load image via URL
            } catch (IOException ex) {
               ex.printStackTrace();
            }
         }
      }
      imgWidth = imgFrames[0].getWidth(null);
      imgHeight = imgFrames[0].getHeight(null);
   }

   /** Update the position based on speed and direction of the sprite */
   public void update() {
      x += speed * Math.cos(Math.toRadians(direction));  // x-position
      if (x >= CANVAS_WIDTH) {
         x -= CANVAS_WIDTH;
      } else if (x < 0) {
         x += CANVAS_WIDTH;
      }
      y += speed * Math.sin(Math.toRadians(direction));  // y-position
      if (y >= CANVAS_HEIGHT) {
         y -= CANVAS_HEIGHT;
      } else if (y < 0) {
         y += CANVAS_HEIGHT;
      }
      direction += rotationSpeed;  // update direction based on rotational speed
      if (direction >= 360) {
         direction -= 360;
      } else if (direction < 0) {
         direction += 360;
      }
      ++currentFrame;    // display next frame
      if (currentFrame >= imgFrames.length) {
         currentFrame = 0;
      }
   }

   /** Custom painting codes on this JPanel */
   @Override
   public void paintComponent(Graphics g) {
      super.paintComponent(g);  // paint background
      setBackground(Color.WHITE);
      Graphics2D g2d = (Graphics2D) g;

      transform.setToIdentity();
      // The origin is initially set at the top-left corner of the image.
      // Move the center of the image to (x, y).
      transform.translate(x - imgWidth / 2, y - imgHeight / 2);
      // Rotate about the center of the image
      transform.rotate(Math.toRadians(direction),
            imgWidth / 2, imgHeight / 2);
      // Apply the transform to the image and draw
      g2d.drawImage(imgFrames[currentFrame], transform, null);
   }

   /** The Entry main method */
   public static void main(String[] args) {
      // Run the GUI codes on the Event-Dispatching thread for thread safety
      SwingUtilities.invokeLater(new Runnable() {
         @Override
         public void run() {
            JFrame frame = new JFrame(TITLE);
            frame.setContentPane(new AnimatedFramesDemo());
            frame.pack();
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.setLocationRelativeTo(null); // center the application window
            frame.setVisible(true);
         }
      });
   }
}
```

Dissecting the Program

[TODO]

Code Example 2: Frames Organized in a Stripe

In this example, all the frames of an animated sequence are kept in a single file organized in rows and columns.



```
import javax.imageio.ImageIO;
import java.net.URL;
import java.awt.*;
import javax.swing.*;
import java.io.*;

/** Animating image frames. All frames kept in a stripe. */
@SuppressWarnings("serial")
public class AnimatedFramesInStripe extends JPanel {
   // Named-constants
   static final int CANVAS_WIDTH = 640;
   static final int CANVAS_HEIGHT = 480;
   public static final String TITLE = "Animated Frame Demo";

   private String imgFilename = "images/GhostStripe.png";
   private int numRows, numCols, numFrames;
   private Image img;         // for the entire image stripe
   private int currentFrame;  // current frame number
   private int frameRate = 5; // frame rate in frames per second
   private int imgWidth, imgHeight;  // width and height of the image
   private double x = 100.0, y = 80.0; // (x,y) of the center of image
   private double speed = 8;           // displacement in pixels per move
   private double direction = 0;       // in degrees
   private double rotationSpeed = 1;   // in degrees per move

   /** Constructor to set up the GUI components */
   public AnimatedFramesInStripe() {
      // Setup animation
      loadImage(imgFilename, 2, 4);
      Thread animationThread = new Thread () {
         @Override
         public void run() {
            while (true) {
               update();   // update the position and image
               repaint();  // Refresh the display
               try {
                  Thread.sleep(1000 / frameRate); // delay and yield to other threads
```

```java
               } catch (InterruptedException ex) { }
            }
         }
      };
      animationThread.start();  // start the thread to run animation

      // Setup GUI
      setPreferredSize(new Dimension(CANVAS_WIDTH, CANVAS_HEIGHT));
   }

   /** Helper method to load image. All frames have the same height and width */
   private void loadImage(String imgFileName, int numRows, int numCols) {
      URL imgUrl = getClass().getClassLoader().getResource(imgFileName);
      if (imgUrl == null) {
         System.err.println("Couldn't find file: " + imgFileName);
      } else {
         try {
            img = ImageIO.read(imgUrl);  // load image via URL
         } catch (IOException ex) {
            ex.printStackTrace();
         }
      }
      numFrames = numRows * numCols;
      this.imgHeight = img.getHeight(null) / numRows;
      this.imgWidth = img.getWidth(null) / numCols;
      this.numRows = numRows;
      this.numCols = numCols;
      currentFrame = 0;
   }

   /** Returns the top-left x-coordinate of the given frame number. */
   private int getcurrentFrameX() {
      return (currentFrame % numCols) * imgWidth;
   }

   /** Returns the top-left y-coordinate of the given frame number. */
   private int getCurrentFrameY() {
      return (currentFrame / numCols) * imgHeight;
   }

   /** Update the position based on speed and direction of the sprite */
   public void update() {
      x += speed * Math.cos(Math.toRadians(direction));  // x-position
      if (x >= CANVAS_WIDTH) {
         x -= CANVAS_WIDTH;
      } else if (x < 0) {
         x += CANVAS_WIDTH;
      }
      y += speed * Math.sin(Math.toRadians(direction));  // y-position
      if (y >= CANVAS_HEIGHT) {
         y -= CANVAS_HEIGHT;
      } else if (y < 0) {
         y += CANVAS_HEIGHT;
      }
      direction += rotationSpeed;  // update direction based on rotational speed
      if (direction >= 360) {
         direction -= 360;
      } else if (direction < 0) {
         direction += 360;
      }
      ++currentFrame;    // displays next frame
      if (currentFrame >= numFrames) {
         currentFrame = 0;
      }
   }

   /** Custom painting codes on this JPanel */
   @Override
   public void paintComponent(Graphics g) {
      super.paintComponent(g); // paint background
      setBackground(Color.WHITE);
      Graphics2D g2d = (Graphics2D) g;

      int frameX = getcurrentFrameX();
      int frameY = getCurrentFrameY();
      g2d.drawImage(img,
            (int)x - imgWidth / 2, (int)y - imgHeight / 2,
            (int)x + imgWidth / 2, (int)y + imgHeight / 2,
            frameX, frameY, frameX + imgWidth, frameY + imgHeight, null);
   }

   /** The Entry main method */
   public static void main(String[] args) {
      // Run the GUI codes on the Event-Dispatching thread for thread safety
      SwingUtilities.invokeLater(new Runnable() {
         @Override
         public void run() {
            JFrame frame = new JFrame(TITLE);
            frame.setContentPane(new AnimatedFramesInStripe());
```

```
                frame.pack();
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.setLocationRelativeTo(null); // center the application window
               frame.setVisible(true);
            }
         });
      }
   }
```

Dissecting the Program

[TODO]

4. High Performance Graphics

4.1 Full-Screen Display Mode (JDK 1.4)

Reference: Java Tutorial's "Full-Screen Exclusive Mode API" @http://docs.oracle.com/javase/tutorial/extra/fullscreen/index.html.

You could check if full-screen mode is supported in your graphics environment by invoking `isFullScreenSupported()` of the screen `GraphicsDevice`:

```
GraphicsEnvironment env = GraphicsEnvironment.getLocalGraphicsEnvironment();
GraphicsDevice defaultScreen = env.getDefaultScreenDevice();
System.out.println("isFullScreenSupported: " + defaultScreen.isFullScreenSupported());

// Enter fullscreen mode
setUndecorated(true);
setResizable(false);
defaultScreen.setFullScreenWindow(this);  // "this" JFrame
```

To enter fullscreen mode, use `GraphicsDevice`'s `setFullScreenWindow(JFrame)`. To leave the fullscreen mode and return to windowed mode, use `setFullScreenWindow(null)`. You should not try to `setSize()` or resize the window in full screen mode.

You could run your program in fullscreen (without the window's title bar) by invoking `JFrame`'s `setUndecorated(true)`.

There are a few ways to find out the current screen size:

```
// via the default Toolkit
Dimension dim = Toolkit.getDefaultToolkit().getScreenSize();
int screenWidth = (int)dim.getWidth();
int screenHeight = (int)dim.getHeight();
```

Code Example 1: Running in Fullscreen Mode

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

/** Testing the full-screen mode */
@SuppressWarnings("serial")
public class FullScreenDemo extends JFrame {

   /** Constructor to set up the GUI components */
   public FullScreenDemo() {
      // Check if full screen mode supported?
      GraphicsEnvironment env = GraphicsEnvironment.getLocalGraphicsEnvironment();
      GraphicsDevice defaultScreen = env.getDefaultScreenDevice();
      if (!defaultScreen.isFullScreenSupported()) {
         System.err.println("Full Screen mode is not supported!");
         System.exit(1);
      }

      // Use ESC key to quit
      addKeyListener(new KeyAdapter() {
         @Override
         public void keyPressed(KeyEvent e) {
            if (e.getKeyCode() == KeyEvent.VK_ESCAPE) {
               System.exit(0);
            }
         }
      });

      setContentPane(new DrawCanvas());
      setDefaultCloseOperation(EXIT_ON_CLOSE);
      setUndecorated(true);
      setResizable(false);
      defaultScreen.setFullScreenWindow(this); // full-screen mode
      setVisible(true);
   }

   /** DrawCanvas (inner class) is a JPanel used for custom drawing */
   private class DrawCanvas extends JPanel {
      @Override
      public void paintComponent(Graphics g) {
         super.paintComponent(g);
```

```java
         setBackground(Color.BLACK);

         // Paint messages
         g.setColor(Color.YELLOW);
         g.setFont(new Font(Font.DIALOG, Font.BOLD, 30));
         FontMetrics fm = g.getFontMetrics();
         String msg = "In Full-Screen mode";
         int msgWidth = fm.stringWidth(msg);
         int msgAscent = fm.getAscent();
         int msgX = getWidth() / 2 - msgWidth / 2;
         int msgY = getHeight() / 2 + msgAscent / 2;
         g.drawString(msg, msgX, msgY);

         g.setColor(Color.WHITE);
         g.setFont(new Font(Font.DIALOG, Font.PLAIN, 18));
         fm = g.getFontMetrics();
         msg = "Press ESC to quit";
         msgWidth = fm.stringWidth(msg);
         int msgHeight = fm.getHeight();
         msgX = getWidth() / 2 - msgWidth / 2;
         msgY += msgHeight * 1.5;
         g.drawString(msg, msgX, msgY);
      }
   }

   /** The Entry main method */
   public static void main(String[] args) {
      // Run the GUI codes on the Event-Dispatching thread for thread safety
      SwingUtilities.invokeLater(new Runnable() {
         @Override
         public void run() {
            new FullScreenDemo();
         }
      });
   }
}
```

Code Example 2: Switching between Fullscreen and Windowed Mode

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

/** Testing the full-screen mode */
@SuppressWarnings("serial")
public class FullScreenEscDemo extends JFrame {
   // Windowed mode settings
   private static String winModeTitle =
         "Switching between Full Screen Mode and Windowed Mode Demo";
   private static int winModeX, winModeY;          // top-left corner (x, y)
   private static int winModeWidth, winModeHeight; // width and height

   private boolean inFullScreenMode;    // in fullscreen or windowed mode?
   private boolean fullScreenSupported; // is fullscreen supported?

   /** Constructor to set up the GUI components */
   public FullScreenEscDemo() {
      // Get the screen width and height
      Dimension dim = Toolkit.getDefaultToolkit().getScreenSize();
      // Set the windowed mode initial width and height to about fullscreen
      winModeWidth = (int)dim.getWidth();
      winModeHeight = (int)dim.getHeight() - 35; // minus task bar
      winModeX = 0;
      winModeY = 0;

      // Check if full screen mode supported?
      GraphicsEnvironment env = GraphicsEnvironment.getLocalGraphicsEnvironment();
      final GraphicsDevice defaultScreen = env.getDefaultScreenDevice();
      fullScreenSupported = defaultScreen.isFullScreenSupported();

      if (fullScreenSupported) {
         setUndecorated(true);
         setResizable(false);
         defaultScreen.setFullScreenWindow(this); // full-screen mode
         inFullScreenMode = true;
      } else {
         setUndecorated(false);
         setResizable(true);
         defaultScreen.setFullScreenWindow(null); // windowed mode
         setBounds(winModeX, winModeY, winModeWidth, winModeHeight);
         inFullScreenMode = false;
      }

      // Use ESC key to switch between Windowed and fullscreen modes
      this.addKeyListener(new KeyAdapter() {
         @Override
         public void keyPressed(KeyEvent e) {
            if (e.getKeyCode() == KeyEvent.VK_SPACE) {
               if (fullScreenSupported) {
                  if (!inFullScreenMode) {
                     // Switch to fullscreen mode
                     setVisible(false);
                     setResizable(false);
                     dispose();
                     setUndecorated(true);
                     defaultScreen.setFullScreenWindow(FullScreenEscDemo.this);
                     setVisible(true);
                  } else {
                     // Switch to windowed mode
                     setVisible(false);
                     dispose();
                     setUndecorated(false);
                     setResizable(true);
                     defaultScreen.setFullScreenWindow(null);
                     setBounds(winModeX, winModeY, winModeWidth, winModeHeight);
                     setVisible(true);
                  }
                  inFullScreenMode = !inFullScreenMode;
                  repaint();
               }
            } else if (e.getKeyCode() == KeyEvent.VK_ESCAPE) {
               System.exit(0);
            }
         }
      });

      // To save the window width and height if the window has been resized.
      this.addComponentListener(new ComponentAdapter() {
         @Override
         public void componentMoved(ComponentEvent e) {
            if (!inFullScreenMode) {
               winModeX = getX();
               winModeY = getY();
            }
         }

         @Override
         public void componentResized(ComponentEvent e) {
            if (!inFullScreenMode) {
               winModeWidth = getWidth();
               winModeHeight = getHeight();
            }
         }
      });

      setContentPane(new DrawCanvas());
      setDefaultCloseOperation(EXIT_ON_CLOSE);
      setTitle(winModeTitle);
      setVisible(true);
   }

   /** DrawCanvas (inner class) is a JPanel used for custom drawing */
   private class DrawCanvas extends JPanel {
      @Override
      public void paintComponent(Graphics g) {
         super.paintComponent(g);
         setBackground(Color.BLACK);

         // Draw a box to indicate the borders
         Graphics2D g2d = (Graphics2D)g;
         g2d.setStroke(new BasicStroke(8));
         g2d.setColor(Color.RED);
         g2d.drawRect(0, 0, getWidth()-1, getHeight()-1);

         // Paint messages
         g.setColor(Color.YELLOW);
         g.setFont(new Font(Font.DIALOG, Font.BOLD, 30));
         FontMetrics fm = g.getFontMetrics();
         String msg = inFullScreenMode ? "In Full-Screen mode" : "In Windowed mode";
         int msgWidth = fm.stringWidth(msg);
         int msgAscent = fm.getAscent();
         int msgX = getWidth() / 2 - msgWidth / 2;
         int msgY = getHeight() / 2 + msgAscent / 2;
         g.drawString(msg, msgX, msgY);

         g.setColor(Color.WHITE);
         g.setFont(new Font(Font.DIALOG, Font.PLAIN, 18));
         fm = g.getFontMetrics();
         msg = "Press SPACE to toggle between Full-screen/windowed modes, ESC to quit.";
         msgWidth = fm.stringWidth(msg);
         int msgHeight = fm.getHeight();
         msgX = getWidth() / 2 - msgWidth / 2;
         msgY += msgHeight * 1.5;
         g.drawString(msg, msgX, msgY);
      }
   }

   /** The Entry main method */
   public static void main(String[] args) {
      // Run the GUI codes on the Event-Dispatching thread for thread safety
      SwingUtilities.invokeLater(new Runnable() {
         @Override
         public void run() {
            new FullScreenEscDemo();
         }
      });
   }
}
```

4.2 Rendering to the Display & Double Buffering

The common problems in rendering a graphic object or image to the display are:

- Flashing (or flickering): caused by clearing the display and then drawing the graphics.
- Image Tearing: For a moving object, the user sees part of the new image and part of the old one.

The common way to resolve these display rendering problem is via so-called *double buffering*.

A few techniques are available in Java for double buffering:

- BufferStrategy (JDK 1.4)
- BufferredImage
- other??

[TODO]

4.3 Splash Screen

To show a splash screen before launching your application, include command-line VM argument "`-splash:`*`splashImagefilename`*" to display the image.

To show a progress bar over the splash screen, need to write some codes to overlay the progress bar on top of the splash screen. The following shows a simulation:

```
import java.awt.*;

/** Splash Screen Demo (with a Progress Bar)
    Run with VM command-line option -splash:splashImageFilename */
public class SplashScreenDemo {

   public static void main(String[] args) {
      SplashScreen splash = SplashScreen.getSplashScreen();
      if (splash == null) {
         System.err.println("Splash Screen not available!");
      } else {
         // Okay, Splash screen created
         Dimension splashBounds = splash.getSize();
         Graphics2D g2d = splash.createGraphics();

         // Simulate a progress bar
         for (int i = 0; i < 100; i += 5) {
            g2d.setColor(Color.RED);
            g2d.fillRect(0, splashBounds.height / 2,
                  splashBounds.width * i / 100, 20);
            splash.update();
            try {
               Thread.sleep(200); // Some delays
            } catch (Exception e) {}
         }
         g2d.dispose();
         splash.close();
      }
   }
}
```

[TODO] Can we use the `JProgressBar` class?

REFERENCES & RESOURCES

- Java 2D Tutorial @ http://docs.oracle.com/javase/tutorial/2d/TOC.html.
- Java Tutorial's "Full-Screen Exclusive Mode API" @ http://docs.oracle.com/javase/tutorial/extra/fullscreen/index.html.
- Jonathan S. Harbour, "Beginning Java 5 Game Programming".

Latest version tested: JDK 1.7.2
Last modified: April, 2012