Lecture I
BASIC JAVA 2D

Graphical User Interfaces (GUI) is a universal paradigm for interacting with computers today. GUI elements include windows, scroll bars, radio buttons, text input windows, etc. GUI is studied under the topic of Human Computer Interfaces (HCI). We view GUI's purely as a tool towards our goals in visualization. In particular, we need to understand the GUI model of Java.

The GUI model of Java comes in two flavors. The original model is called **Abstract Windowing Toolkit** or AWT for short. A newer model, since Java 1.2, is **Swing Toolkit**. There is a perception that Swing replaces AWT. This is not exactly true. Here is one way to see their connection: AWT has two subsystems, a graphics imaging subsystem and a user-interface subsystem. Both were limited in their ability to achieve sophisticated design goals. Swing is really the replacement for the user-interface subsystem. We also see where **Java 2D** comes into this landscape: it is the replacement for the graphics imaging subsystem of AWT.

§1. Overview of Java 2D

REFERENCES:

[1] Java 2D Graphics, J.Knudsen, O'Reilly

[2] Java2D, Hardy

[3] Core Web Programming, M.Hall and L.Brown, Sun.

It is part of the core classes for Java 2 Platform (formerly JDK 1.2). So you can immediately access Java 2D on the Java 2 Platform. The original 2D graphics API is found in AWT, in the following packages: `java.awt, java.awt.image.` +In Java2D, 5 new packages are available: `java.awt.color, font, geom, print, image.renderable` +The last package (`image.renderable`) is a bridge to Java Advanced Imaging API (JAI), which we will not treat. There is also `com.sun.image.codec.jpeg`. Although not an official part of Java, it is included in JDK1.3.

The Abstract Windowing Toolkit (AWT) in JDK1.0 and 1.1 can be split into two parts: the User Interface (UI) and the drawing toolkit. In Java 2, the UI became Swing, and the drawing toolkit became Java2D. Java2D is part of JFC and part of Java Media APIs. Java2D API can draw on AWT components and Swing components.

Since it is part of core Java 2, there is no special package to download (except possibly for `com.sun.image.codec.jpeg`). The starting point for doing Java2D graphics is the `Graphics2D` class. Hence you must import this package:

```
import java.awt.Graphics2D;  +
```

To produce any graphical image, you call methods of the `Graphics2D` class. So the first step is to get a `Graphics2D` object. Below we indicate how to obtain such an object. For backward compatibility, a `Graphics` object from the original java.awt can be cast into a Graphics2D object.

A Graphics2D object stores a set of **attributes objects**. The states of these objects are collectively known as the Graphics2D's **rendering context**. There are 7 attributes: pen style, fill style, compositing style, transform, clip, font and rendering hints. The corresponding methods to set them are **setStroke**, **setPaint**, **setComposite**, **setTransform**, **setClip**, **setFont**, **setRenderingHints**.

Note that Graphics2D holds references to the attribute objects, not clones. This means if you change an attribute object during rendering, you must call the appropriate set methods (above) to inform Graphics2D; otherwise you get unpredictable behavior.

After setting up the rendering context, you are ready to produce render graphical images. Graphics2D can render three kinds of data: (1) Geometric Shapes (e.g., lines, rectangles, polygons) (2) Text (a string of characters) (3) Raster Images.

Graphics2D has four **rendering methods**:
(a) draw: renders the outline of a geometric shape, following the stroke or paint attributes.
(b) fill: renders a geometric shape by filling it with color or pattern specified by the paint attribute.
(c) drawString: renders a text string, following on the font and paint attributes.
(d) drawImage: renders an image.

Note that (a) draw and (b) fill are for geometric shapes. For backward compatibility and convenience, the analogous rendering methods from the original `Graphics` are available: drawX and fillX (where X = Oval, Arc, Rect) are available. This is convenient because you can get these basic shapes in a single instruction. But remember that only integer coordinates are supported by these methods.

In the simplest case, suppose you want to render geometric shapes in a `Component` (AWT) or a `JComponent` (Swing). Each `Component` is automatically given `Graphics` object. Moreover, the `paint(Graphics g)` method of the component is given this object as argument. In this case, you simply derive a `Graphics2D` object from this `Graphics` object, as part of your override of the `paint(g)` method:

```
    public void paint(Graphics g)  // (1) Get Graphics2D object: Graphics2D g2 =
(Graphics2D) g; // (2) Set Rendering Context g2.setPaint(Color.yellow); // paint
attribute // (3) Render Geometry int x=50; int y=50; int w=100; int h=60; g2.fillRect(x,
y, w, h); // draws solid yellow rect // (4) Render Text g2.setPaint(Color.blue); // reset
paint attribute g2.drawString("Hello World!", 55, 80);   +
```

This code is seen in the `HelloWorld.java` program.

§2. AWT Overview

The HelloWorld.java+ program in this directory uses a frame component to display all the GUI elements are derived from the Component class.

AWT defines the `Component` class, from which all windows and GUI controls are derived. E.g., an Applet is a window type, and a radio button is a GUI control. Thus, Applets and radio buttons are derived from Components. All Components has the `paint()` method (see [3] p.473 for complete list).

Some Components can contain other Components. E.g., a Panel Component. Such Components are derived from the Container class which is in turn derived from the Component class (of course). One method of Container classes is `add(Component c)`. (see [3] p.485 for complete list).

There are 8 main window types: components are classified as standalone or otherwise. For simple applications, the standalone `Frame` Component is the only one you need (see ApplicationFrame below).

```
    * Frame: Container and standalone. It has a title, menu bar, border, cursor, and
icon image. Uses: starting point for graphics, and for pop-up windows in Applets. *
Canvas: simplest, non-container, non-standalone. Uses: drawing area, custom component
(e.g., an image label) * Panel: borderless, container, non-standalone. Uses: grouping and
organizing other components. * Applet: container, standalone. Uses: embedding in html *
ScrollPane: borderless, holds ONLY one Component (but this Component can be a Container).
Uses: to hold something that may be too big for a window. * Dialog: Stripped-down frame:
no cursor, menu or icon image. Uses: To freeze other AWT components until it is closed.
E.g., to confirm quiting. * FileDialog: Uses: to save or load files. * Window: Frames are
derived from Windows. Standalone. Uses: popup window with no border or title bar.  +
```

```
    * Button: can be pushed and held down * Checkbox: can be checked * Radio buttons:
this is a group of Checkboxes in which at most one can be checked * Choice Menus: * List
Box: * TextField: * TextArea; * Label: * Scrollbars and Sliders: * Pop-up Menus:  +
```

The above discusses AWT. Current books tend to emphasize Swing. So what is their difference? AWT relies on "peer-based" rendering to achieve platform independence. But it is difficult to achieve a consistent look-and-feel this way, giving rise to platform-dependent bugs. Swing avoids these problems by using a non-peer-based approach. Thus it is able to faithfully reproduce the look-and-feel of each platform (Windows, Motif, etc), it allows programs to specify the look-and-feel. It also has a new look-and-feel, called "Metal". Two problems with Swing are (1) it is not fully supported by all browsers and hence is less reliable on the web, and (2) it may be slower than AWT. See [3], chapter 13. Hence most of our examples of Java2D is based AWT rather than Swing. We will briefly mention how to accomplish the same with Swing.

§3. Rendering Through a Frame

Above, we illustrated what is perhaps most typical method of grabbing a Graphics2D object: from a component. Another method is through an Applet (but this is really a derived Component, but more later).

Let us go through the process in detail. You need to first decide on a Component to display the Graphics2D object. Usually, you need to set up a Component for the screen, which in turn must be placed in some Container with Listeners activated. The simplest Component to use is the Frame which is standalone (see above).

The following `ApplicationFrame` class from Knudsen (p.8) is useful for displaying a Graphics2D object. Most of our basic examples will use it by subclassing it and overriding its `paint()` method.

```
// file: ApplicationFrame.java import java.awt.*; import java.awt.event.*;

public class ApplicationFrame extends Frame

// Constructor with default title: public ApplicationFrame()  this("ApplicationFrame
v1.0");

// Constructor with specified title: public ApplicationFrame(String title)
super(title); createUI(); //ApplicationFrame

// General setup utility: protected void createUI()  setSize(500, 400); center();

// Our frame generates events -- so we need a Listener. // The new WindowAdapter here
is made the Listener, // and we override its windowClosing(e) method:

addWindowListener(new WindowAdapter()  public void windowClosing(WindowEvent e)  //
NOTE: addWindowListener() is a method of java.awt.Window dispose(); System.exit(0);  //
The only non-null method of new WindowAdapter is "windowClosing" // which listens for the
window closing event! ); //createUI

// Center the frame in the screen // -- (1) get the screen dimensions, and
the window dimensions // -- (2) compute the position (x,y) of window if it is
centered // -- (3) set at this location public void center()  Dimension screenSize =
Toolkit.getDefaultToolkit().getScreenSize(); Dimension frameSize = getSize(); int x =
(screenSize.width - frameSize.width) / 2; int y = (screenSize.height - frameSize.height)
```

```
/ 2; setLocation(x, y); // the window is placed at (x,y) //center

   // Main method public static void main(String[] args)  // main has only one frame:
class myFrame = new ApplicationFrame("myFrame") // myFrame has only one method:
public void paint(Graphics g) Graphics2D g2 = (Graphics2D)g; // set rendering context:
g2.setPaint(Color.blue); // rendering g2.drawString("This text in BLUE", 10, 20); //
again: g2.setPaint(Color.red); Line2D.Double ld = new Line2D.Double(10, 25, 100, 150);
g2.draw(ld); // draw a diagonal //paint //new ApplicationFrame //main //class  +
```

§4. Rendering Images

We now show a second method of getting a Graphics2D object, from an image. Reference [2, chap.11].

A Graphics2D object is device-dependent: for instance, when obtained from a Component, it knows about how to draw on the screen. When obtained from an image, it knows how to display that image. The concept of an `Image` in the original AWT is device-independent, and basically stores a raster image. The Java2D counterpart of Image is called a `BufferedImage`, found in the package `java.awt.image.BufferedImage`. This package is critical for more advanced image processing of Java2D. Unlike its parent `java.awt.Image` which has no easy access to the image data, BufferedImage provides a complete set of methods to manipulate image bits.

What is the relation between Graphics2D and BufferedImage? Well, a BufferedImage object has the method `createGraphics()` to return an associated Graphics2D object, which in turn can be used to draw or manipulate the raster image. This process is illustrated in the next program.

We use the `Utilities` class from [1, p.199]. For our present purposes, we focus on the method `makeBufferedImage(Image im)` which converts a Image to a BufferedImage.

```
     //file: Utilities.java import java.awt.*; import java.awt.event.*; import
java.awt.image.BufferedImage; import java.net.URL;

   public class Utilities  private static final Component sComponent = new Component() ;
private static final MediaTracker sTracker = new MediaTracker(sComponent); private static
int SID = 0;

   public static boolean waitForImage(Image image)  int id; synchronized (sComponent)
id = sID++;  sTracker.addImage(image id); try  sTracker.waitForID(id);  catch
(InteruptedException ie)  return false;  if (sTracker.isErrorID(id)) return false; return
true; // waitForImage

   public static Image blockingLoad(String path)  Image image = Toolkit.getDefaultToolkit().getImage(pa
if (waitForImage(image) == false) return null; return image; //blockingLoad

   public static BufferedImage makeBufferedImage(Image image)  //makeBufferedImage   +
```

§5. REVIEW: Steps in using Graphics2D

We have already seen two specific uses of Graphics2D objects. We summarize the four basic steps:
(1) Acquire of Graphics2D object. E.g., from a Component.
(2) Set up attributes of the "graphic context". E.g., font attributes.
(3) Select the "graphic object" to render. E.g., text.
(4) Apply the appropriate rendering operation. E.g., drawString.

We elaborate on the 4 steps:

(1) This is the only device-dependent step. There are 3 main devices: monitor, image buffer and printer. We have already illustrated the first two devices. For each, you acquire the corresponding Graphics2D object differently:

(a) Monitor device: typically, you acquire this my constructing a GUI component such as Panel (AWT) or JFrame (Swing). The method `paint(g)` in such an object is automatically given a graphics object when it is invoked by the system. RECALL that you do not call `paint(g)` directly! For backwards compatibility to AWT, `g` is a Graphics object. So in Java2D, you first cast `g` into a Graphics2D object: `paint(Graphics g) Graphics2D g2 = (Graphics2D)g; ...` +Alternatively, `java.awt.Component.getGraphics()` will return such an object.

(b) Printer device: if P is a printable object (analogous to Component), then P has a `paint(g)` method which takes a Graphics object `g` as input parameter. By implementing `P.paint(g)`, you can use this Graphics(2D) object.

(c) Offscreen Image Buffer: There are getGraphics() and createGraphics() methods for such buffers. The method is `public abstract Image getImage(String filename) public abstract Image getImage(URL url)` +

To avoid implementing this abstract method, the simplest solution is to obtain a Toolkit subclass thus:

```
Image myImage = Toolkit.getDefaultToolkit().getImage("image.gif");  +
```

Alternatively: use the createImage() method, which avoids caching [p.182, Knudsen].

```
public abstract Image createImage(URL url) public abstract Image createImage(String
filename) public Image createImage(byte[] imagedata) public abstract Image
createImage(byte[] imagedata, int offset, int length)  +
```

If you want BufferedImage bI from an Image iM, you first construct a BufferedImage object bI (of some height and width, etc). Then, invoke bI.createGraphics() to return a Graphics2D object g2. Finally, use Image in iM to draw on bI. This is illustrated in the following method

```
public static BufferedImage makeBufferedImage(Image im)  +
```

which is in the `Utilities` class defined in [2, p.198]. See below.

(2) Attributes of the graphic context are classified under stroke, paint, composite, transform, clip, font,

hints.

(3) Graphics2D objects can render three kinds of objects:
(a) geometric forms (under java.awt.Shape),
(b) text, and
(c) images (under java.awt.image.BufferedImage)

(4) The rendering method depends on the object to be rendered. Suppose s is a Shape object (see below). To draw s in red color on the Graphics2D object g2, we simply do:

```
s.setPaint(color.red); // get paint color s.draw(s);  +
```

If you want to render a gray text, you proceed thus:

```
s.setPaint(color.grey); s.drawString("This is in Gray");  +
```

§6. Geometric Shapes

Above we used common geometric shapes such as rectangles and lines. We now consider the general situation.

We begin with the most basic geometric object, the point. In AWT, you have the `Point` class. In Java2D, this becomes the class `Point2D`. It is an abstract class, but it has 2 inner classes, Point2D.Double and Point2D.Float which have concrete implementations! Note that Point objects have integer coordinates, in contrast to `Point2D.Float` or `Point2D.Double`. The methods of Point2D are: `setLocation(x,y)`, `getX()`, `getY()`, `distance(P)`.

This pattern of abstract class having inner classes which provide implementations will be repeated for the other geometric objects (Lines, QuadCurves, etc).

Points are not directly renderable: only shapes are renderable, and points are used in describing shapes.

A shape is mainly defined by a `PathIterator`. A path iterator is simply a sequence of "segment drawing instructions". There are 5 segment instructions:

- `public void moveTo(x, y)`: This sets the current position at (x,y).
- `public void lineTo(x, y)`: This draws a straint line segment from the current position to (x,y). The new current position is (x,y).
- `public void quadTo(x1, y1, x2, y2)`: This draws a quadratic curve segment from the current position to (x2,y2), and the new current position is (x2, y2). The quadratic curve is controlled by the control points (x1,y1).

- `public void curveTo(x1, y1, x2, y2, x3, y3)`: This draws a cubic curve segment from the current position to (x3,y3), and the new current position is (x3, y3). The cubic curve is controlled by the control points (x1,y1) and (x2,y2).
- `public void closePath()`: This closes the path with a straight line segment. If you need quadratic or cubic curve, you must do it yourself.

Shapes are either Areas or GeneralPath. Areas are regions of the plane and Boolean operators are defined on such areas. GeneralPath includes the following special cases: Line2D, QuadCurve2D, CubicCurve2D, RectangularShape. Finally, RectangularShape has Rectangle2D, RoundRectangle2D, Arc2D, Ellipse2D as subclasses.

The simplest curve is a straight line segment, which is simply (but misleadingly) called `Line2D` in Java2D. As with Point2D, this is an abstract class with inner classes `Line2D.Float` and `Line2D.Double`.

While AWT's know about specific shapes such as rectangles, circles, etc, Java2D deals with a generalization called a `Shape`. The main determinant of a Shape is a path comprisng path segments which have one of 5 types: SEG_MOVETO, SEG_LINETO, SEG_QUADTO, SEG_CUBICTO, SEG_CLOSE. This path is represented by the class `PathIterator`, where the iterator terminology is familiar from other areas of programming: the idea of an iterator is that you can traverse a list of items, have the notion of a current position, ability to position yourself at the starting position, moving to the next position, and checking if you have reached the end position. It is clear that this will be very useful for drawing the path. To obtain the PathIterator of a Shape `s`, we call the method `s.getPathIterator()`.

For instance, the shape `Rectangle2D(0, 0, x, y)` has the PathIterator that the amounts to the following: `move to 0, 0 line to x, 0 line to x, y line to 0, y line to 0, 0 close +`

There are facilities to construct a construct a `GeneralPath` which has properties of a Shape (e.g., you can call a Graphics2D object to draw it). You contruct a GeneralPath by adding one path segment at a time.

See DragKing.java (p. 43, Knudsen), reproduced in this directory. You will see a line, quadCurve and cubicCurve, with their control points and control tangents. You can interactively click on any control point and drag it. The curve would continuously change as you drag. Great demo.

§7. Rendering Text

END OF LECTURE