Standard Java Graphics

Standard Java graphics are a bit weird. All drawing on the screen results from implicit or explicit calls of `repaint()` method, which posts an event on the event queue and eventually results in the paint method being called.

Below we explain how to pop up a window and to create and draw geometric objects in the window. We then see what extra must be done to draw images loaded from a file. Finally we show how to do simple animations using a timer to trigger redrawing of the images.

Creating and Drawing Geometric Objects

Here are some details that can best be understood by looking at the program MyGraphicsDemo.

1. The classes defined below are found in packages `java.awt` and `java.awt.geom`. Thus you will need to include statements to import `java.awt.*` and `java.awt.geom.*`.

2. `paint(g)` is called automatically when a window is created or uncovered in a window. It can also be forced to execute by sending a `repaint()` message to a component. That method schedules a call to an `update(g)` method as soon as possible.

3. The inherited `update(g)` method erases the component and then calls `paint(g)`. The Graphics object `g` for the component is automatically provided by the system.

4. The user overrides `paint(g)` to actually draw something on the component. All drawing commands are either in this method or in a method called from `paint`.

5. The `Graphics` object should be thought of like a pen that is responsible for doing the actual drawing. The newer Java graphics actually use a class called `Graphics2D`, which extends `Graphics`, and that is the one actually passed to the `paint` method. Unfortunately, for compatibility with old code, the parameter type is still `Graphics`. To use the newer method described below, you must begin by casting the graphics context to `Graphics2D`:

   ```
   /**
    * Draw figures on the window that has graphics g
    * @param g - the graphics context of the current window
    */
   public void paint(Graphics g) {
      Graphics2D g2 = (Graphics2D)g;
      g2.draw(...);
      ...
   }
   ```

   The `draw` message is sent to the `Graphics2D` object to draw various framed graphics objects, as shown in the sample code. The method `fill` is used to draw filled objects. The message `drawString` is used to write a string on the screen. The message `setPaint(aColor)` changes the color of the pen. It stays that color until changed again.

6. The graphics objects can be used with either type `float` or `double` for coordinates and dimensions. We will always use the double versions. The graphics classes that you should use are thus

   - `Rectangle2D.Double`

- `RoundRectangle2D.Double`
- `Ellipse2D.Double`
- `Arc2D.Double`
- `Line2D.Double`

Their rectangle constructor takes the following parameters:

```
myRect = new Rectangle2D.Double(x, y, width, height);
```

The others are similar. Details can be found in the javadoc documentation for the Java 5classes, which may be found from a link on the Documents and Handouts page of the course web page. For those of you used to the `objectdraw` library, you should note that there is no canvas parameter. Note also that this creates the object, but does not actually draw it anywhere.

If `g2` is a `Graphics2D` object, then you can actually draw the object using one of the two following methods:

```
g2.draw(myRect);
g2.fill(myRect);
```

The first draws a framed version of the object, while the second draws a filled version.

Every time repaint is called or a window is uncovered the `paint` method will be executed. As a result, you should make sure that everything that you want to appear on the screen will be redrawn by the `paint` method.

7. You can draw on virtually any component in a window, including a `JApplet`, `JFrame`, or `JPanel`. Normally in this class we will be drawing on a `JFrame` or `JPanel`.

   Suppose you just want to pop up a window and write on its surface. The kind of window we will be using in Java is called a `JFrame`.[1] A program that just pops up a window and draws something on it will have the following form:

```
public class GraphicsExample extends JFrame {
    private static final int WINDOW_WIDTH = ...;
    private static final int WINDOW_HEIGHT = ...;

    // instance variable declarations

    /**
     * Create the window
     * @ param title -- the text to be show on the title bar
     */
    public GraphicsExample(String title) {
       super(title);
       ...
```

[1] Yes, there is a `JWindow`, but it doesn't have a title bar or any of the other gadgets that we associate with modern computer windows.

```
    }

    /**
      * Draw figures on the window that has graphics g
      * Called by repaint() or whenever the screen needs to be
      * refreshed
      * @param g - the graphics context of the current window
      */
    public paint(Graphics g) {
       Graphics2D g2 = (Graphics) g;
       // commands using g2
    }

    // Create the window of the desired size and display it.
    public static void main(String[] args) {
       GraphicsExample f = new GraphicsExample("StdGraphicsDemo");
       f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
       f.setSize(WINDOW_WIDTH, WINDOW_HEIGHT);
       f.setVisible(true);
    }
}
```

The class extends `JFrame`, so when it is created it will pop up a window. The call to the super constructor in the constructor for `GraphicsExample` ensures that the string in title shows in the title bar of the window. The paint method casts the graphics context to be of type `Graphics2D` so the above commands will work.

The `main` method constructs an object of the class, which will result in a window being popped up. The next three commands make sure that clicking in the red "go-away box" at the top of the window will result in the program terminating, set the window size at whatever values are held in the constants, and make the window visible. When the window is made visible for the first time, the repaint method will automatically be called, which will erase the window and eventually call `paint` with the graphics context of the window.

Creating and Drawing Images

Often we will want to import images into our program and draw them in a window. This takes a bit more work, and needs the following extra code.

1. You first need to bring the image from a file into computer memory. Unfortunately it can take a (relatively) long time to bring an image from memory (e.g., a `jpeg` or `gif` file), so after we request the item to be brought in, we may need to wait for it. I suggest you use the following method that I have written to make sure that the image is fully loaded before your program proceeds.

```
/**
 * Retrieve an image from file "filename", return the
 * Image when it has loaded completely.
 * @param the name of the file containing the image
```

```
 * @return the loaded Image.
 */
private Image getCompletedImage(String fileName) {
   Toolkit toolkit = Toolkit.getDefaultToolkit();
   Image myImage = toolkit.getImage(fileName);
   MediaTracker mediaTracker = new MediaTracker(this);
   mediaTracker.addImage(myImage, 0);
   try {
      mediaTracker.waitForID(0);
   } catch (InterruptedException ie) {
      System.out.println(ie);
      System.exit(1);
   }
   return myImage;
}
```

Roughly, the method retrieves a default toolkit that has a method that will allow reading in an image in a file with name given by `fileName`. After the image is fetched with the `getImage` method, a `MediaTracker` object is created. The image is added to the media tracker object, which then waits for the loading of the image to be complete. If it fails to complete then an exception will be thrown and the program will exit. Otherwise, when the image is done loading, it will be returned from the method.

You can try to do without the `mediaTracker` object, but if you try to access info about the image (e.g., ask its width or height), you might not get accurate information.

2. Using the `getCompletedImage` method, an image can be loaded and displayed in a window with the following code:

```
Image myImage = getCompletedImage("filename.jpg");
g2.drawImage(flowerImage, x, y, this);
```

where "filename.jpg" should be replaced by the name of the file holding the image. This code should be placed in a `paint(g)` method or in a method called from the `paint` method.

Creating simple animations

Animations take a bit of work, as they require a separate thread to do the updating of the picture that is different from the thread that is actually doing the drawing and responding to other events. The following is a description of a low overhead way of doing animation that relies on events fired by a timer rather than creating a separate thread (which we will talk about later in the term). See the program SimpleTimerAnimation for a simple example of an application that creates an animation using a `Timer` object.

1. In the constructor for the main class (the one that extends `JFrame`), insert commands of the form

```
Timer myTimer = new Timer(interval,this);
myTimer.start();
```

This creates a timer that will generate an `ActionEvent` every `interval` milliseconds (which is 1/1000 second). The timer can be turned off by sending it a `stop()` message.

2. The constructor given above creates a timer that expects the object that created the timer to respond to the timer events. To be notified that the timer has gone off the class must implement the interface `ActionListener`. Therefore the header of the class should look like:

```
public class MyClass extends JFrame implements ActionListener {
```

   The interface `ActionListener` has a single method:

```
public void actionPerformed(ActionEvent evt);
```

   that the class must implement.

3. Each time the timer is fired (using whatever interval was chosen when the timer was created), the method `actionPerformed` will be called. The value of the formal parameter `evt` will be the timer event that triggered the call. It is rarely needed in the method body, so you may ignore it for now, though it must appear as a formal parameter of the method. The body of the method should include whatever is necessary to update the state of all of the objects and then call `repaint()`.

4. As before, `repaint()` will end up calling `paint(Graphics g)`. That routine is responsible for doing all of the actual `draw` or `fill` commands. Of course that method can itself call draw routines for objects that are in your program. For example, in a program with several gardens, the `paint` method can send a `draw(Graphics g)` message to each of the gardens, where the corresponding method draws all of the plants in the garden.

5. In order to clear the screen between invocations of `paint`, I found it necessary to call `super.paint(g)` at the beginning of the `paint` method. I believe this is necessary so that all the subcomponents also get repainted properly. When this was not done, the earlier images were not erased.