PHAS3459: Scientific Programming Using Object Oriented Languages

Module 9: Graphics and Applets

Dr. B. Waugh and Dr. S. Jolly
Department of Physics & Astronomy, UCL

Contents

1. Aims of Module 9

This module introduces graphics, graphical user interfaces (GUIs) and applets. After this module you should be able to:

- create simple graphical user interfaces;
- use some common techiques to produce graphical displays;
- write simple applets that can be displayed in a web browser.

2. Graphics and GUIs

The classes and methods we have looked at so far have been able to interact with the outside world, including human users of the programs, only by means of text. We have seen how to read text from the keyboard, from files and from web servers; we have also seen how to print text to the screen or to a file. However, this is not always the best way to display information or to accept input. In many cases we would like to display images on a screen, or let the user use a mouse to tell the computer what to do.

We do not have time to cover all aspects of using graphics in Java, but this module will provide an introduction to the subject and illustrate some of the techniques that can be applied and extended in creating more complex graphical applications. We will also look at an *applet*: a special kind of Java program that can be run within a suitably equipped web browser. Applets provide a way of adding more interactive functions, graphics and animations to a web site than are possible in simple HTML pages, although parts of this functionality are slowly being superseded by CSS and HTML5 and above. They were widely used before other technologies such as Flash become available, and are probably one of the reasons the Java language has become so widely known and used.

3. Using the Swing Toolkit

The Swing toolkit, provided as part of the standard Java distribution, provides a variety of components that can be put together to display graphical information and create Graphical User Interfaces (GUIs) that can interact with the user in a variety of ways. Swing builds on the earlier AWT (Abstract Window Toolkit) package, providing added functionality and making it easier to build applications. We will largely be using classes from the Swing (`javax.swing`) package, but will also need to import some from AWT (`java.awt`).

3.1 A Simple Swing Application

In order to illustrate the basic structure of a program that uses Swing components, we will start with a completely non-interactive example that simply displays a string in a window. As this is such a simple program we will use only one class, but we will separate the actual creation and set-up of the window into a separate method instead of putting everything in `main` since this will make it easier to adapt and extend this program to create more complex ones.

```
import javax.swing.*;

/**
 * Simple Swing application using JLabel to display text.
 */
public class TextDisplay {

  /** Create and display a JFrame containing a JLabel. */
  private static void createAndDisplayGui() {
    JFrame frame = new JFrame("Swing example");
    // Exit application if window is closed
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    JLabel label = new JLabel("Hello Java programmers!");
    frame.add(label);       // Add label to frame
    frame.pack();           // Set component sizes and layout
    frame.setVisible(true); // Display the resulting frame
  }

  /** Call method to create and display GUI. */
  public static void main(String[] args) {
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        createAndDisplayGui();
      }
    });
  }
}
```

There are two Swing *components* used in this example. `JFrame` is a top-level container representing a window on the computer screen: a *container* is a component that can contain other components, and this one is *top-level* because it is not contained within any other container. `JLabel` is one of the simplest Swing components: it simply contains a text string.

By default, even if the `JFrame` is closed, the application will continue running, and using resources on the computer, even if it is not actually doing anything. We avoid this by calling `setDefaultCloseOperation` with the argument `JFrame.EXIT_ON_CLOSE` so that if the window is closed the application will exit. Other possibilities include `JFrame.DISPOSE_ON_CLOSE` which simply disposes of the `JFrame` object when the window is closed, which may be desirable if the application has other windows and should continue running.

The call to `pack` arranges the components (only one in this case) within the window, and sets the size of the window to be just big enough to contain the components. Nevertheless, by default it is still possible to resize the window by dragging its edges and corners with the mouse.

There are some further features of the `main` method that may be puzzling at first glance. Rather than simply calling `createAndDisplayGui`, `main` creates an instance of an *anonymous inner class* that implements the `Runnable` interface and calls `createAndDisplayGui` from its `run` method. This `Runnable` object is then passed to the method `SwingUtilities.invokeLater`. This is done because the classes of the Swing toolkit are not *thread-safe*: they cannot be guaranteed to function correctly if they are accessed from more than one thread. The way this is resolved is by having a single thread,

the *event-dispatching thread*, that handles all calls to Swing methods. The `invokeLater` method adds the `Runnable` object to a queue of requests on the event-dispatching thread, where its `run` method will be called in its turn, possibly after other requests have been handled, and usually after the `invokeLater` method has returned.

3.2 Creating Graphics

The next example illustrates how we can start to create images on the screen using the methods of the class `java.awt.Graphics`. We start with another Swing component, `JPanel`, a generic container that we can use to contain other components or, as in this case, extend to create our own custom panel type. This time we will separate the application into two classes, one representing the panel:

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

/** JPanel containing some lines and text. */
public class LinesPanel extends JPanel {

  /** Constructor just sets size of panel. */
  public LinesPanel(int width, int height) {
    setPreferredSize(new Dimension(width,height));
  }

  /**
   * Must override this method, which is called
   * by the Swing framework whenever the display
   * needs updating.
   */
  protected void paintComponent(Graphics g) {
    super.paintComponent(g); // call superclass method first
    int width = getWidth();
    int height = getHeight();
    for (double r=0.0; r<1.0; r+=0.05) {
      int x = (int) (width*r);
      int y = height - (int) (height*r);
      g.drawLine(x, 0, 0, y);
    }
    Font f = new Font("TimesRoman",Font.BOLD,28);
    g.setFont(f);
    g.drawString("Some lines",width/2,height/2);
  }
}
```

and another containing the main program, which creates a `LinesPanel` and adds it to a `JFrame`:

```
import javax.swing.*;

/** Simple Swing application illustrating graphics. */
public class Lines extends JPanel {

  /** Create and display a JFrame containing a LinesPanel. */
  private static void createAndDisplayGui() {
    JFrame frame = new JFrame("Swing graphics example");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    LinesPanel panel = new LinesPanel(400,400);
    frame.add(panel);       // Add panel to frame
    frame.pack();           // Set component sizes and layout
    frame.setVisible(true); // Display the resulting frame
  }

  /** Call method to create and display GUI. */
  public static void main(String[] args) {
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        createAndDisplayGui();
      }
    });
  }
}
```

The main program is almost identical to that in the earlier example, with the difference being in the component (still just one) that is added to the `JFrame`.

In extending the `JPanel` class, we override the method `paintComponent`, which is automatically called by Swing whenever the display of the component needs to be updated, such as when the window is resized. Apart from that, all we need is a contructor that performs any necessary initialization, in this case just setting the size of the panel.

The `paintComponent` method is passed an argument of type `Graphics` representing the graphics *context* belonging to this component, which enables various drawing operations to be performed on it. In this example we draw some lines and a string. See the API JavaDoc web pages for details of these and the other available methods.

3.3 User Input

So far we have been displaying information as text or graphics on the screen, but to create a graphical *user interface* (GUI) clearly we need ways of making our application respond to the actions of the user. One simple input method is the *dialog box*, which can easily being created using the `JOptionPane` class. Here the `showInputDialog` method causes a dialog box to be displayed on the screen, and returns the user's input once this has been entered:

```
String input = JOptionPane.showInputDialog("Enter an integer");
try {
  int num = Integer.parseInt(input);
  System.out.format("%d squared is %d",num,num*num);
}
catch (NumberFormatException e) {
  System.out.println("Not an integer!");
}
```

This provides a means of getting text entered by the user in applications where there is no command line, but it is still rather limited. Often we want a GUI to have multiple controls (buttons, text fields, sliders etc.) and we want it to respond to each one as the user manipulates it. We do not know where the next input will come from, so we cannot simply do nothing as we wait for a particlar response. This is where we have to dive into *event-driven programming*, as opposed to the *thread-driven programming* we have encountered so far. Rather than orchestrating sequences of actions by calling methods in turn, we provide methods to deal with particular events and let a software framework such as Swing call each one when it is needed.

There are many types of *control* provided by Swing, of which we use only a few here. The rest can be found in the JavaDoc of the package `javax.swing` The general approach is to instantiate a control, add it to a container, and give it an `EventListener` object providing methods that will be called whenever the control has any input to report.

In the following example we use the classes `JTextField` and `JButton` to obtain user input. The former will generate an `ActionEvent` object when the *enter* key is pressed while the control is active, while the latter will generate an `ActionEvent` when it is clicked. We create an object that implements the `ActionListener` interface and register it with the control using its `addActionListener` method. The `actionPerformed` method of our listener will then be called when any input arrives, so we ensure that this method carries out any action that is needed in response to the input.

This is quite simple as long as the action can be completed quickly. Since all events are handled on one thread, the event-dispatching thread, taking too long to respond can cause the entire application to become unresponsive. In such a case another thread should be used to take on the time-consuming task. This can be done using the methods described in the first part of this module, but in Java 5 and 6 there are more sophisticated multi-threading tools available such as the `Executor` interface and the `SwingWorker` class. In the examples presented here none of the tasks will take long enough to warrant this treatment.

This is the first example where we are adding more than a single component to a container, and we therefore need to tell the container how its child components should be laid out. This is done using a `LayoutManager` implementation. Here we use a `BoxLayout` with the `BoxLayout.Y_AXIS` orientation, meaning components will be arranged vertically with each appearing below the previous component added.

The following class can be instantiated and displayed using a main program with the same structure as that seen in the previous examples:

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

/** Panel illustrating text and button input. */
public class TextInputPanel extends JPanel implements ActionListener {
  // These components are accessed from more than
  // one method so they must be member variables.
  private JLabel label;
  private JTextField text;
  private JButton button;

  /** Create panel containing the required components. */
  public TextInputPanel() {
    // Simple layout with vertical arrangement of components.
    setLayout(new BoxLayout(this,BoxLayout.Y_AXIS));
    label = new JLabel("Please enter some text.");
    text = new JTextField(20);
    button = new JButton("Reverse");
    add(label);   // add components to this frame
    add(text);
    add(button);

    // Call actionPerformed method of this object
    // if text is entered in field.
    text.addActionListener(this);
    button.addActionListener(this); // Likewise if button clicked.
  }

  /**
   * This method is called by Swing whenever
   * an action is detected in the JTextField
   * or JButton. It is required by the
   * ActionListener interface.
   */
  public void actionPerformed(ActionEvent e) {
    if (e.getSource()==text) {         // action is from JTextField
      label.setText(text.getText()); // copy text from input to label
    }
    else if (e.getSource()==button) {  // action is from JButton
      label.setText(reverse(label.getText()));
    }
  }

  /** Private utility method to reverse a string. */
  private static String reverse(String input) {
    StringBuilder s = new StringBuilder();
    for (int i=input.length()-1; i>=0; --i) {
      s.append(input.charAt(i));
    }
    return s.toString();
  }
}
```

We can respond to movements or clicks of the mouse anywhere in the GUI, not just clicked buttons, using the `MouseListener` interface and implementing methods such as `mouseClicked` as here:

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

/** Panel illustrating mouse input. */
  public class MouseCoordinatesPanel extends JPanel
                               implements MouseListener {

  /** Just need to register this object as MouseListener. */
  public MouseCoordinatesPanel() {addMouseListener(this);}

  /** Print number of mouse button in location of click. */
  public void mouseClicked(MouseEvent e) {
    int iButton = e.getButton();
    int x = e.getX(); int y = e.getY();
    String s = String.format("%d",iButton);
    this.getGraphics().drawString(s, x, y);
  }

  /**
   * We have to implement all the methods of the
   * MouseListener interface, even if they don't
   * do anything. An alternative would be to extend
   * the abstract MouseAdapter class, which provides
   * dummy implementations, but we cannot do this
   * as well as extending JPanel, so we would need
   * to create an additional class.
   */
  public void mousePressed(MouseEvent e) {}
  public void mouseReleased(MouseEvent e) {}
  public void mouseEntered(MouseEvent e) {}
  public void mouseExited(MouseEvent e) {}
}
```

Try running this and clicking at various locations in the window using different mouse buttons.

3.4 Animation

Animation creates the illusion of movement by redrawing an image repeatedly with a small change each time. If this is done frequently enough, we perceive the image as being in continuous motion rather than being a succession of still images. It is probably easiest to understand how to do animation by looking at an example. Here we will create a simple applet that will rotate a red square on a blue background.

We start by defining a class to represent a panel on which the rotating shape will be drawn, with few member variables that will be needed. We make our class implement `ActionListener` since we will be generating our animation by generating a regular sequence of events using the class `javax.swing.Timer`. Note that this is *not* the class `java.util.Timer`.

```
/**
 * JPanel containing a rotating square
 * that can be stopped and started.
 */
public class AnimationPanel extends JPanel implements ActionListener {
  private Polygon shape;        // shape to be displayed
  private final int delay = 50; // delay in ms between steps
  private final double delta;   // angle to rotate in each step
  private double angle = 0.0;   // current angle of shape on screen
  private Timer animationTimer; // timer controlling frame rate
}
```

The constructor will set the size of the panel from arguments that are passed to it, create the shape that will be rotated, and set up the `Timer` object that will generate an `ActionEvent` each time the next frame should be drawn:

```
  /**
   * Create panel with rotating shape.
   * @param width width of panel
   * @param height height of panel
   * @param rotationTime time for complete rotation [seconds]
   */
  AnimationPanel(int width, int height, double rotationTime) {
    setPreferredSize(new Dimension(width,height));
    int size = Math.min(width, height) / 4;
    int[] xpts = {size,-size,-size,size};
    int[] ypts = {size, size,-size,-size};
    shape = new Polygon(xpts,ypts,4);
    delta = 2*Math.PI*delay/(rotationTime*1000);
    animationTimer = new Timer(delay,this);
    animationTimer.start();
  }
```

As for our stationary graphics earlier, we need to override `paintComponent` to define how the shape should be drawn:

```
  /** Paint shape at appropriate angle. */
  protected void paintComponent(Graphics g) {
    super.paintComponent(g);
    int height = getHeight();
    int width = getWidth();
    // Fill in background
    g.setColor(Color.BLUE);
    g.fillRect(0, 0, width, height);
    // Now move origin to centre of panel
    g.translate(width/2, height/2);
    // Rotate and draw shape
    g.setColor(Color.RED);
    Polygon rotatedShape = rotatePolygon(shape, angle);
    g.fillPolygon(rotatedShape);
  }
```

This makes use of a private method to rotate the shape:

```
/**
 * Private utility method to rotate the polygon.
 * @param poly polygon to be rotated
 * @param angle angle in radians by which to rotate polygon
 * @return rotated polygon
 */
private static Polygon rotatePolygon(Polygon poly, double angle) {
  Polygon newPoly = new Polygon();
  for (int i = 0; i < poly.npoints; i++) {
    double x = poly.xpoints[i]*Math.cos(angle)+
    poly.ypoints[i]*Math.sin(angle);
    double y = poly.ypoints[i]*Math.cos(angle)-
    poly.xpoints[i]*Math.sin(angle);
    newPoly.addPoint((int) x, (int) y);
  }
  return newPoly;
}
```

The `actionPerformed` method will be called each time the animation timer generates an event. It increases the angle of the shape and tells Swing to repaint the component. We have seen that `paintComponent` is called automatically when Swing detects a relevant change such as a resizing of the window. When the change is due to an internal change of state, as here, we call `repaint`, which will cause `updateComponent` to be called along with other necessary actions.

```
/**
 * This is called by the animation Timer object
 * at regular intervals to rotate the shape and
 * update the display.
 */
public void actionPerformed(ActionEvent event) {
  angle += delta;
  repaint();
}
```

Finally we provide some methods that enable the animation to be stopped and started:

```
/** Start the animation */
public void start() {animationTimer.start();}

/** Stop the animation */
public void stop() {animationTimer.stop();}
```

A second class defines a panel that contains the first along with a row of buttons to start and stop the animation and to exit the application:

```
/**
 * Rotating square animation applet with
 * start, stop and exit buttons.
 */
public class AnimationGuiPanel extends JPanel
                               implements ActionListener {
  private AnimationPanel animPanel; // panel containing animation
  private JButton startButton;
  private JButton stopButton;
  private JButton exitButton;

  /** Create JPanel containing animation panel and buttons. */
  public AnimationGuiPanel() {
    super();
    setPreferredSize(new Dimension(250,300));
    setLayout(new BoxLayout(this,BoxLayout.Y_AXIS));

    animPanel = new AnimationPanel(200,200,10.0);
    startButton = new JButton("Start");
    stopButton  = new JButton("Stop");
    exitButton = new JButton("Exit");

    startButton.addActionListener(this);
    stopButton.addActionListener(this);
    exitButton.addActionListener(this);

    JPanel buttonPanel = new JPanel();
    buttonPanel.setLayout(new BoxLayout(
                     buttonPanel,BoxLayout.X_AXIS));
    buttonPanel.add(startButton);
    buttonPanel.add(stopButton);
    buttonPanel.add(exitButton);

    add(animPanel);
    add(buttonPanel);
  }

  /** Respond to button clicks */
  public void actionPerformed(ActionEvent e) {
    if (e.getSource()==startButton) start();
    else if (e.getSource()==stopButton) stop();
    else if (e.getSource()==exitButton) System.exit(0);
  }

  /** Start animation when applet is started */
  public void start() {animPanel.start();}

  /** Stop animation when applet is stopped */
  public void stop() {animPanel.stop();}
}
```

The main program once more is quite simple:

```
/**
 * Rotating square animation applet with
 * start, stop and exit buttons.
 */
public class Animation {
  /** Create and display JFrame containing animation GUI panel */
  public static void main(String[] args) {
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        JFrame frame = new JFrame("Animation demo");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(250,300);
        JPanel panel = new AnimationGuiPanel();
        frame.add(panel);
        frame.setVisible(true);
      }
    });
  }
}
```

4. Applets

An applet differs from the programs or *applications* you have written so far in that it does not have a `main` method. Rather than running as a self-contained program, it provides methods that are called by the web browser or applet viewer.

An applet must extend the class `java.applet.Applet` or, if it uses Swing components, `javax.swing.JApplet`, which provides simple versions of all the necessary methods. In order to do anything interesting it must then override some or all of these methods.

An applet is also subject to some security restrictions that do not apply to applications. The fact that an applet can run in a web browser makes it easy and convenient to run for any user who can access the web server, but could also open up opportunities for a malicious programmer to interfere with the user's computer. For this reason several restrictions are imposed. For example, an applet is not normally allowed to read or write files on the computer that is executing it, although some browsers give the user the option of allowing such access. An applet cannot make network connections except to the server it was loaded from and cannot communicate with applets loaded from a different server.

4.1 Lifecycle of an Applet

The methods that are called by the browser include:

- `init`: called when the applet is initially loaded. This is used to carry out the initialization of the applet and is called once when the applet is loaded. It fills a similar role to the constructor in other classes.
- `start`: called when the browser decides to start running the applet, after `init` has been called. Anything that the applet does after initialization must be started off either by the `start` method or by a user carrying out an action such as clicking with a mouse. Often one or more threads are started in this method.
- `stop`: called when the browser decides to stop running the applet. This may happen, depending on the browser, when the user moves to a different web page.
- `destroy`: called when the browser unloads the applet. Most applets do not need to override this method, but it can be used to release resources where this is not done in the `stop` method.

Generally the `init` and `destroy` methods are called once, while `start` and `stop` may be called several times, for example if the user moves away from the page containing the applet and then returns to it. However, not all browsers behave in the same way.

A Swing applet has access to all the same components that an application can use, but these are contained within a `JApplet` instead of a `JFrame` as the top-level container. Another difference is the use of the `init` method to carry out the initialization of the components. The `start` and `stop` methods can be used to give the browser the option of not using CPU time to run the applet when it is not on display. Finally, the `init` method must not return until the applet has been fully initialized, so rather than using `invokeLater` we must use the related `invokeAndWait` method in `SwingUtilities` and catch the exceptions it declares.

4.2 Creating an Applet

We can create an applet using the same `AnimationPanel` and `AnimationGuiPanel` classes we created above. This example was structured in a sufficiently flexible way that we can simply replace the main `Animation` program with the following `AnimationApplet` class:

```
/**
 * Rotating square animation applet with
 * start and stop buttons.
 */
public class AnimationApplet extends JApplet {
  private AnimationGuiPanel panel;

  /** Create animation GUI panel and add to applet */
  public void init() {
    try {
      SwingUtilities.invokeAndWait(new Runnable() {
        public void run() {
          panel = new AnimationGuiPanel();
          add(panel);
        }
      });
    }
    catch (Exception e) {
      System.err.println("Unable to initialize applet");
    }
  }

  /** Start animation when applet is started. */
  public void start() {panel.start();}

  /** Stop animation when applet is stopped. */
  public void stop() {panel.stop();}
}
```

Eclipse will give you a warning: “The serializable class TestApplet does not declare a static final SerialVersionUID field of type long”. This does not stop the applet from working, but you can get rid of the error by adding the following member variable:

```
  private static final long serialVersionUID = 0;
```

Eclipse will run this for you using its built-in applet viewer. You may need to change the size of the viewer for this applet: select “Run configurations...” from the “Run” menu; select the applet class; select the “Parameters” tab; set the width and height.

One change that you might want to make to the `AnimationGuiPanel` class if using it in an applet is to remove the “Exit” button, which has no effect when running within a web browser.

4.3 Including an Applet in a Web Page

You may want to try including an applet in a web page of your own, so we will give brief instructions here. If you want to put your page with its applet on the UCL web server, you should look at the instructions at:

http://www.ucl.ac.uk/isd/services/website-apps/databases/personal-webpages

These will tell you where to put your files and how to make them viewable. The following procedure should, however, work on any web server to which you have access. Because it is the browser and not the server that runs the Java applet, there is no need for any special features on the server.

In a directory on the web server you will need to create an HTML file containing an `applet` tag giving the location of your applet file. To create this file you can use a text editor like WordPad or a web development program such as Dreamweaver.

```
<html>
 <head><title>Java applet demo</title></head>
 <body>
  <applet code="module9/AnimationApplet.class"
                        width="400" height="450">
  Your browser does not recognize the applet tag.
  </applet>
 </body>
</html>
```

You will also need to copy the byte-code (`.class`) file for the applet, and any other classes that it uses, into the appropriate location on the web server. Note that each class file must be placed in a subdirectory with the same name as the Java package in which you have placed the class in Eclipse. For example, assuming you have an applet class called `AnimationApplet` in the package `module9`, you might call the HTML file `animation.html` and put it in the folder `html.pub\javademo`, and copy the class files `AnimationApplet.class`, `AnimationPanel.class` etc. to the folder `html.pub\javademo\module9`.

Unfortunately not all browsers have a recent version of Java installed, but for example Firefox on Desktop@UCL does have Java version 7, so you should be able to view your applets this way.

If you need to make your applet run in a browser that only has Java version 1.4, and you cannot update the browser's Java plugin, you will need to compile your applet for the older Java version. You can only change this setting for an entire project in Eclipse, and you need at least version 5.0 for some of the code in earlier modules. One way round this is to create a separate project in Eclipse for your applets, called something like `PHAS3459Applets`.

To do this, go to the “File” menu in Eclipse, go to “New” and then select “Java Project” from the drop-down menu. Give your project a name and click “Finish”. Select the new project in the Project Explorer and then select “Properties” from the “Project” menu. Select “Java Compiler” in the left-hand menu and tick the box “Enable project specific settings”.

You should then select “1.4” in the pull-down menu labelled “Compiler compliance level”, as shown in Fig. 9.1.

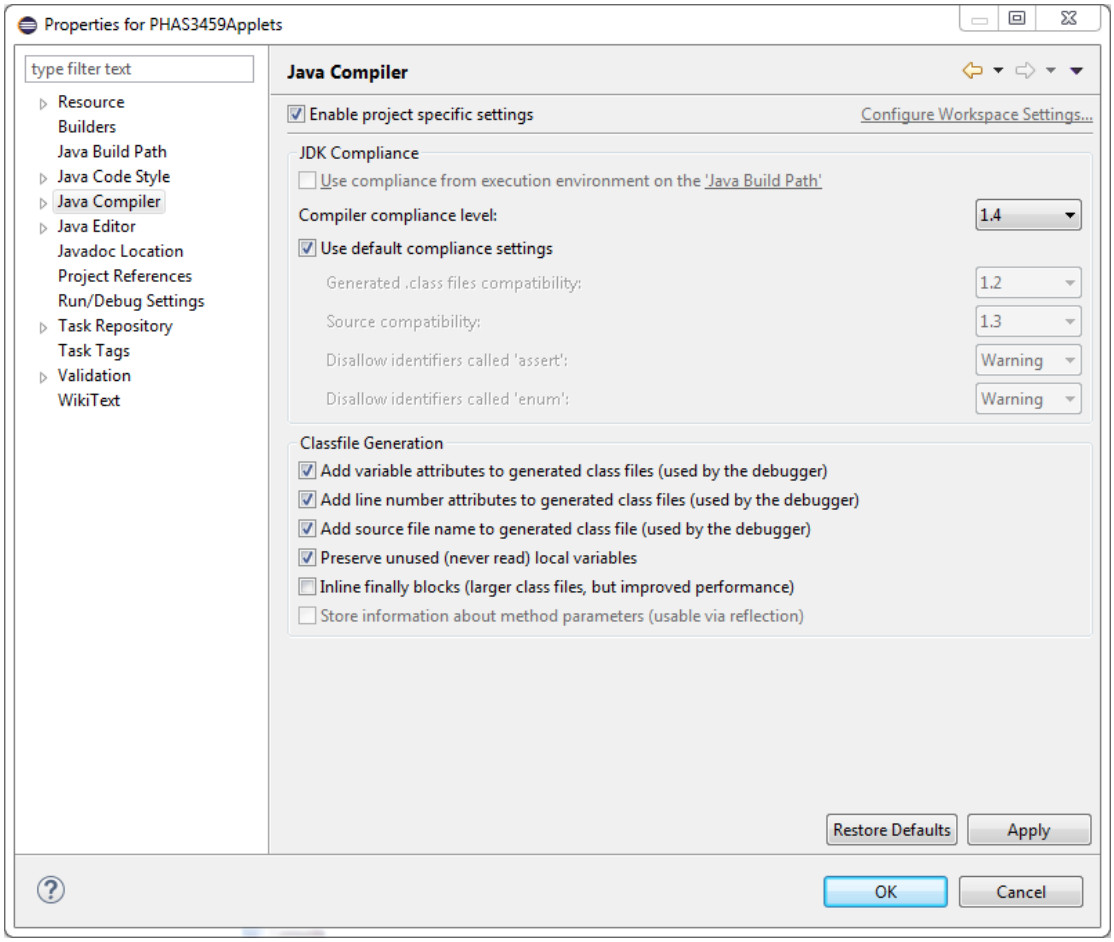

Figure 9.1: Changing the compiler compliance level for compatibility with version 1.4 of the Java Runtime Environment.

5. Summary

This module has only covered the basics of graphics and applets. There are many books devoted to these areas, and many web pages offering guidance, tutorials and examples. Some of these are better, and some more up to date than others, but you should now be in a position to find other sources of information and build on your existing knowledge if you want to develop your skills further. There is a selection of links to applet examples and tutorials in the *Resources* page linked from the course home page.