Workbook 7

Introduction

Last week you built your own graphical interface with Java Swing in order to make your program more accessible to other users. Unfortunately, whilst your interface looked pretty, it wasn't exactly functional! This week you will be adding code to make your application usable.

This is the last Programming in Java Workbook, but not the last class; there is another practical class next week. If you have enjoyed the course you might be interested in taking part in an Undergraduate Research Opportunities Programme (UROP) run by the Programming in Java Lecturers. The programme runs for ten weeks over the summer and this year will involve writing Java programs for Android mobile phones. All participants are paid a stipend to cover living costs; last year this was £2,200. Further details are available here: http://www.cl.cam.ac.uk/research/dtg/summer/

Important

You may find Sun's Swing Tutorial helpful:

http://java.sun.com/docs/books/tutorial/uiswing/

Remember to check the course website regularly for announcements and errata:

http://www.cl.cam.ac.uk/teaching/current/ProgJava

Handling events in Swing

Almost all Graphical User Interface (GUI) libraries (of which Swing is just one) operate in an event-oriented manner. This entails a mode of operation in which your program starts up and enters an *event loop* which services an *event queue*. Events are added to the event queue in response to the mouse being clicked or the window needing to be redrawn. The event loop consists of taking the next event from the queue and handling it and sleeping if there are no further events to handle. In Swing the event loop (and the background thread which handles it) are managed for you behind the scenes.

You have already written code which interacts with the event loop. One example is the

```
protected paintComponent(Graphics g)
```

method which you overrode in `GamePanel`. This method is called on a component in response to a paint event (a request that the window be redrawn). Similarly, you have also written code which creates events. One example is the call to `repaint()` which you wrote in the display method in `GamePanel`. Calling `repaint()` does not actually do any painting. Instead, it simply inserts a paint event into the event queue. This event is eventually serviced by the event loop which calls the `paintComponent` method and the drawing gets done.

There are two things to take away from this. Firstly, things get done in a GUI program by raising events and handling events. Secondly, anything you do in response to an event shouldn't take too long to execute. If it does then the event loop will spend all its time running your program and so won't be able to handle events like drawing the window. This is a common problem which you have most likely seen before in which a program seems to freeze up and fails to draw its window properly. This isn't the fault of the operating system, or the GUI library, its the fault of the software developer who wrote the program.

Below is a simple program which demonstrates event handling in Swing. Read the code carefully in combination with the information in the subsequent paragraphs:

```
package uk.ac.cam.your-crsid.tick7;

import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JButton;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.BoxLayout;

public class HelloActionWorld extends JFrame {

 private JLabel label;

 //an "inner" class which handles events of type "ActionEvent"
 private class ButtonAction implements ActionListener {
  public void actionPerformed(ActionEvent e) {
   label.setText("Button pressed");        //update text shown in "label"
  }
 }

 HelloActionWorld() {
  super("Hello Action");                   //create window & set title text
  setDefaultCloseOperation(EXIT_ON_CLOSE); //close button on window quits app.

  //configure the layout of the pane associated with this window as a "BoxLayout"
  setLayout(new BoxLayout(getContentPane(),BoxLayout.Y_AXIS));

  label = new JLabel("Button unpressed");  //create graphical text label
  add(label);                              //associate "label" with window
  JButton button = new JButton("Press me");//create graphical button
  add(button);                             //associated "button" with window

  //add a new instance of "ButtonAction" as an event handler for "button"
  button.addActionListener(new ButtonAction());

  setSize(320,240);                        //set size of window
 }

 public static void main(String[] args) {
  HelloActionWorld hello = new HelloActionWorld(); //create instance
  hello.setVisible(true);                          //display window to user
 }
}
```

The interesting line here is

```
button.addActionListener(new ButtonAction());
```

This registers a new event handler on the button created above. This means that when the user clicks the mouse on the button, the operating system will pass an event to Swing indicating that this has occurred. This event goes into the event queue. The event loop will eventually collect this event from the queue, work out which component it applies to (the button) and then call the `ActionListeners` (of which there can be many) associated with that button. The `ActionListener` is an interface defined in the Java Standard Library (just like `World` is an interface defined in another library). The `ActionListener` interface proscribes a single method:

```
void actionPerformed(ActionEvent e);
```

Classes which implement this interface must provide an implementation of this method. This method then gets called in response to an event occurring on the component for which the `ActionListener` has been registered.

The `ActionListener` interface is implemented with an *inner class* called `ButtonAction`. Inner classes are defined inside their parent class and have access to all the fields and methods in the parent (even private ones). Inner classes are non-static by default. This means you should think of them as inner-objects. All the *instance* members of the parent object are available in the inner-object. For example, if you look closely at `ButtonAction` class you will see it refers to `label` which is an instance variable of the parent class. If you needed to write the full name for `label` out explicitly you would write `HelloActionWorld.this.label`. If you wrote `this.label` you would have a compile error because `this` refers to the inner-class rather than the parent. It is also possible to make an inner-class static. In this case it cannot refer to any instance members of the parent class.

You could instead define a normal class in a separate file for `ButtonAction`. This is not desirable because the two classes work closely together and so access to private members is useful. It would be possible to pass a *reference* to `label` to another object rather than using an inner class, however, in more complex examples, the inner class might need to access many different parts of the parent class and it would be very tedious to have to pass them all explicitly.

The remainder of the `HelloActionWorld` class is similar to `HelloSwingWorld` that you saw last week, the new bits being the specification of a vertical box layout as the layout manager of the class, the addition of a `JButton` to the graphical interface, and the addition of an instance of `ButtonAction` as the event handler to `button`.

1. Copy the code for `HelloActionWorld` into a new file with a suitable name and directory structure. Compile and run the program and click on the button labelled "Press me".

2. Create a new field of type `int` called `count` inside `ButtonAction` and initialise `count` to zero. Increment `count` every time the button is pressed, and use the value of `count` to update the text of the label to read "`Button pressed` *n* `time(s)`" where *n* is the number of times the button has been pressed.

A further improvement on inner classes are *anonymous inner classes*. These allow us to define the inner class at the point of use (rather than as a class member) and avoid naming it at all. Consider the following example (and try it out if you like):

```
public class AnonTest1 {

  private static class A {
     public void print() { System.out.println("I am A!"); }
  }

  public static void main(String[] args) {
     A instance1 = new A();
     instance1.print();

     A instance2 = new A() {
       public void print() { System.out.println("I am more than A!"); }
     };
     instance2.print();
  }
}
```

This program defines a *static* inner class called `A` which has a single method `print()`. The `main` function creates an instance of `A` and calls `print`. This does the obvious thing. The `main` function then

creates a second instance of A but overrides the `print` method—this is an anonymous inner class. It is still possible to assign our anonymous inner class to a variable of type A because the inner class implicitly *extends* A.

Here is another example which uses a non-static inner class. If you want to try this out remember to put `AnonTest2` and `AnonTest2Run` in separate, appropriately named files.

```
public class AnonTest2 {
  private int counter;

  public class A { // A is visible outside of AnonTest2
    private A() {} // A can only be constructed inside AnonTest2
    public void print() { System.out.println("A: "+counter); }
  }

  public AnonTest2() {
    counter = 0;
  }

  public void incrementCounter() {
    counter++;
  }

  public A getA() {
    A instance1 = new A();
    return instance1;
  }

  public A getSpecialA() {
    A instance2 = new A() {
     public void print() { System.out.println("Special: "+counter); }
    };
    return instance2;
  }
}

public class AnonTest2Run {

  public static void main(String[] args) {
    AnonTest2 anonTest = new AnonTest2();

    AnonTest2.A instance1 = anonTest.getA();
    AnonTest2.A instance2 = anonTest.getSpecialA();

    instance1.print();
    instance2.print();

    anonTest.incrementCounter();

    instance1.print();
    instance2.print();
  }
}
```

This program creates an inner class A which prints out the parent class' counter and also extends A using an anonymous inner class. As a final example here is an alternative form of `HelloActionWorld` with `ButtonAction` rewritten as an anonymous inner class:

```
package uk.ac.cam.your-crsid.tick7;
import javax.swing.JFrame;       import java.awt.event.ActionListener;
import javax.swing.JLabel;       import java.awt.event.ActionEvent;
import javax.swing.JButton;      import javax.swing.BoxLayout;

public class HelloActionWorld2 extends JFrame {
 private JLabel label;

 HelloActionWorld2() {
  super("Hello Action");                  //create window & set title text
  setDefaultCloseOperation(EXIT_ON_CLOSE); //close button on window quits app.
  //configure the layout of the pane associated with this window as a "BoxLayout"
  setLayout(new BoxLayout(getContentPane(),BoxLayout.Y_AXIS));
  label = new JLabel("Button unpressed");  //create graphical text label
  add(label);                             //associate "label" with window
  JButton button = new JButton("Press me");//create graphical button
  add(button);                            //associated "button" with window
  //create an instance of an anonymous inner class to hand the event
  button.addActionListener(new ActionListener(){
   public void actionPerformed(ActionEvent e) {
    label.setText("Button has been pressed");
   }
  });
  setSize(320,240);                       //set size of window
 }

 public static void main(String[] args) {
  HelloActionWorld2 hello = new HelloActionWorld2(); //create instance
  hello.setVisible(true);                          //display window to user
 }
}
```

Notice the phrase "`new ActionListener(){`" which is then followed by any fields and methods which should be specified inside the body of the anonymous inner class; in this case there is a single method `actionPerformed`, but in general there can be multiple methods and fields inside an anonymous inner class, just as you find inside a normal class definition. This anonymous inner class provides an implementation for the interface `ActionListener`; you can also use the anonymous inner class syntax to provide a sub-class of an abstract class and provide implementations of all the abstract methods.

If you wish to refer to a local variable from within the body of an anonymous inner class, then the variable must be be prefixed with the keyword `final`. (The `final` keyword states that this variable can only reference the current object and cannot be changed to reference a different object. The object itself can be changed but this variable can not.) You may reference a field in the containing class from within the anonymous inner class without declaring it to be `final`; the field `label` above is one such example.

3. Copy the code for `HelloActionWorld2` into a new file with a suitable name and directory structure. Compile and run the program and click on the button labelled "Press me".

4. Make the declaration of the field `label` found in `HelloActionWorld2` into a local variable by deleting the field definition and declaring a local variable in the constructor called `label` of type `JLabel`. Compile the modified program. The compiler will complain that `label` is non-final. Make `label` "final" by prefixing the declaration with the keyword `final` and recompile.

5. Create a new field of type `int` called `count` inside the anonymous inner class and initialise `count` to zero. Increment `count` every time the button is pressed, and use the value of `count` to update the text of the label to read "`Button pressed` *n* `time(s)`" where *n* is the number of times the button has been pressed.

Playing a pattern

In the last section you copied and modified a piece of code which contained an event handler class which implemented an `ActionListener`. In this section you will apply the methods you learnt in the previous section to begin to implement event handlers for your graphical version of the `Game of Life`. In order to play a pattern in the Game of Life, you need to update the Game Panel at regular intervals. To do this you should create an instance of the `javax.swing.Timer` class and use the regular events the `Timer` class generates to update the Game Panel. Below are some sample fields and methods which are compatible with the `GuiLife` class you wrote last week:

```
public class GuiLife {

  //...

  private World world;
  private int timeDelay = 500; //delay between updates (millisecs)
  private int timeStep = 0;    //progress by (2 ^ timeStep) each time

  private Timer playTimer = new Timer(timeDelay, new ActionListener() {

   public void actionPerformed(ActionEvent e) {
    doTimeStep();
   }

  });

  void doTimeStep() {
   if (world != null) {
    world = world.nextGeneration(timeStep);
    gamePanel.display(world);
   }
  }

  //...
}
```

In the code above, `playTimer` will generate a new event of type `ActionEvent` every `timeDelay` milliseconds. The anonymous inner class associated with `playTimer` will call the method `doTimeStep` every time an event is fired. The `playTimer` needs to be started before it will start producing events; this can be done by calling the method `start` with no arguments on `playTimer`.

6. Copy across your implementation of `GuiLife` from last week, together with all the classes which `GuiLife` depends on, into the package `uk.ac.cam.your-crsid.tick7`.
7. Include the additional code shown above in the body of the class `GuiLife`; you will need to add suitable import statements for `Timer`, `ActionListener`, and `ActionEvent`.
8. Your implementation of the static main method inside `GuiLife` last week contained a local variable called `w` of type `World`. Delete the local variable `w` and use the field called `world` of type `World` which you added in the previous step instead. (This is required so that the method `doTimeStep` can access a reference to the correct instance of `World`.)
9. Start `playTimer` by calling the `start` method in the static main method just before the call to `setVisible`. (Hint: since `playTimer` is not a static field, you will need to access the field from the instance of `GuiLife` you have already created called `gui`.)
10. Compile and run your code. The Game Panel should now animate the Game of Life.

Step and Speed

In the previous section you animated the Game of Life, but the speed of the animation and the number of steps before the display was updated was fixed to 500 ms and 2^0=1 in the variables `timeDelay` and `timeStep` respectively. In this section you will add event handlers to the widgets representing speed and the number of steps to update these variables dynamically.

There are a variety of different approaches which can be used to connect the event handlers associated with updates from the sliders to `playTimer`. One reasonably neat strategy is to add abstract methods to the Control Panel which are called when the value of the slider is changed, then create an anonymous inner class inside `GuiLife` which implements these abstract methods and updates the `playTimer` object and `timeStep` field. For example, to add support for the speed slider, first make your implementation of Control Panel an abstract class by prefixing the class declaration for `ControlPanel` with the keyword `abstract` and add the following abstract method to the class `ControlPanel`:

```
protected abstract void onSpeedChange(int value);
```

Then call the abstract method from within the body of a `ChangeListener` event handler by placing the following code into the constructor for `ControlPanel` after the initialisation of `speedSlider`:

```
speedSlider.addChangeListener(new ChangeListener() {
public void stateChanged(ChangeEvent e) {
 if (!speedSlider.getValueIsAdjusting())
  onSpeedChange(speedSlider.getValue());
 }
});
```

Finally modify your implementation of `createControlPanel` in `GuiLife` as follows:

```
private JComponent createControlPanel() {
  controlPanel = new ControlPanel(){
   protected void onSpeedChange(int value) {
    playTimer.setDelay(1+(100-value)*10);
   }
  };
  addBorder(controlPanel,Strings.PANEL_CONTROL);
  return controlPanel;
}
```

Notice the body of `createControlPanel` now creates an anonymous inner class which extends `ControlPanel` and provides an implementation for the method `onSpeedChange`. Since this code is inside the body of `GuiLife`, then the method `onSpeedChange` can call `setDelay` on `playTimer` to update the rate at which the Game Panel is updated.

11. Update your implementation of `ControlPanel` and `GuiLife` as suggested above. Run your program and confirm that changing the speed slider does indeed change the speed of playback. (Hint: You'll need to import `ChangeListener` and `ChangeEvent`.)

12. Update your implementation of `ControlPanel` and `GuiLife` so that changes in the step slider will change the number of steps the game board will iterate through before the display is updated. (Hint: you should create a new abstract method called `onStepChange` inside `ControlPanel` and your implementation of it in the anonymous inner class in `GuiLife` should update the value of `timeStep` appropriately.) Run your program and confirm that changing the step slider does indeed change the number of steps specified by the slider before the game board is updated in the Game Panel.

Zoom

Updating the zoom level is very similar to the methodology used earlier for speed and step. The only additional difficulty is the need to update the private field `zoom` which is associated with the `GamePanel` object. This requirement can be addressed by adding a new method called `setZoom` to the `GamePanel` object which takes a single argument of type `int` and returns nothing; the body of the `setZoom` method should update the field `zoom` with the value of the single argument provided to the method. The method `setZoom` can then be called from within `GuiLife` by calling `setZoom` on the field `gamePanel`.

13. Write a method called `setZoom` on the class `GamePanel`. The method should take a single argument of type `int` and return nothing; the body of the `setZoom` method should update the private field `zoom` with the value of the argument passed to the method.

14. Add a `ChangeListener` object to handle events from the `zoomSlider` in `ControlPanel`. Changes in the slider should call an abstract method called `onZoomChange` in the same way as `onSpeedChange` and `onStepChange` were handled earlier.

15. Provide an implementation for the abstract method `onZoomChange` inside the body of an anonymous inner class defined in `GuiLife`. The implementation should call `setZoom` on the field `gamePanel` to change the zoom level of the Game Panel.

16. Update the creation of `zoomSlider` inside `ControlPanel` so that the value of the zoom slider is initially `10`.

Choosing a source location

In this section you will restructure your code to support the functionality defined in the Source Panel. This will be done in five steps:

- Add a method called `getCurrentPattern` to the Pattern Panel to access the current pattern (if any) displayed to the user.
- Add some extra code to the `setPatterns` method in `PatternPanel` to permit an argument with the value `null` indicating that no valid patterns were found.
- Move most of the required "startup" code to initialise variables from the `main` method in `GuiLife` into a new private method called `resetWorld` so that the state of the application can be reset whenever the user selects a new source of input.
- Add appropriate event handlers and abstract methods to `SourcePanel` to update the application whenever the user changes one of the input sources.
- Add an anonymous inner class which extends `SourcePanel` to the method `createSourcePanel` in `GuiLife`, providing an implementation of each of the abstract methods defined in `SourcePanel`.

You will now work through the restructuring in more detail in the rest of this section of the Workbook.

17. Add a private field of type `Pattern` called `currentPattern` to `PatternPanel`.

18. Initialise the value of `currentPattern` to `null` in the constructor for `PatternPanel`.

19. Write a public method called `getCurrentPattern` in `PatternPanel` which takes no arguments and returns a reference of type `Pattern`. The body of `getCurrentPattern` should return the value of the private field `currentPattern`.

20. Add the following code snippet to the *beginning* of the method `setPatterns`:

```
if (list == null) {
 currentPattern = null; //if list is null, then no valid pattern
 guiList.setListData(new String[]{}); //no list item to select
 return;
}
```

21. Add the following code snippet to the *end* of the method `setPatterns`:

```
currentPattern = list.get(0); //select first element in list
guiList.setSelectedIndex(0);  //select first element in guiList
```

22. Compile and run your program to make sure it still works; the first element of the list should now be selected when your program starts.

A new method called `resetWorld` in `GuiLife` should be added to `GuiLife` to provide a generic method of resetting the state of the application if a new source is selected in the Source Panel. The method should be written as follows:

```
private void resetWorld() {
 Pattern current = patternPanel.getCurrentPattern();
 world = null;
 if (current != null) {
  try {
   world = controlPanel.initialiseWorld(current);
  } catch (PatternFormatException e) {
   JOptionPane.showMessageDialog(this,
    "Error initialising world",
    "An error occurred when initialising the world. "+e.getMessage(),
    JOptionPane.ERROR_MESSAGE);
  }
 }
 gamePanel.display(world);
 repaint();
}
```

23. Add the `resetWorld` method as defined above to your implementation of `GuiLife`.

24. Replace the `main` method inside `GuiLife` with the following:

```
  public static void main(String[] args) {
      GuiLife gui = new GuiLife();
      gui.playTimer.start();
      gui.resetWorld();
      gui.setVisible(true);
    }
```

In order to respond to changes in the radio buttons used in the Source Panel, you will need to add event handlers to each of the buttons. You should do this in a similar way to the ones that you've already completed for the previous parts of the user interface, namely add an event handler to `SourcePanel` and from there call an abstract method, then in `GuiLife` create an anonymous inner class which extends `SourcePanel` and provides a suitable implementation. You should also add a new field called `current` of type `JRadioButton` to keep track of the current button the user has selected; this will be useful in the cases where the user selects a new source and that source doesn't work—for example,

if the user selects the "file" radio button, and then presses cancel when asked to select a file, your application should revert the radio button to the one referenced by `current` since the new selection wasn't completed successfully.

For example, in order to add support for the "file" radio button, you should add the following code to the constructor for `SourcePanel`:

```
file.addActionListener(new ActionListener() {
 public void actionPerformed(ActionEvent e) {
  if (setSourceFile())
   //successful: file found and patterns loaded
   current = file;
  else
   //unsuccessful: re-enable previous source choice
   current.setSelected(true);
 }
});
```

and you should add the following abstract method to the class `SourcePanel`:

```
protected abstract boolean setSourceFile();
```

25. Add a private field called `current` of type `JRadioButton` to `SourcePanel` and initialise the field to point to `none` at the end of the constructor.

26. Make the class `SourcePanel` an abstract class and add four protected abstract methods `setSourceNone`, `setSourceFile`, `setSourceLibrary`, and `setSourceThreeStar`; each method should take no arguments and return a `boolean`, just like the example `setSourceFile` shown above.

27. Add four event handlers for the four radio buttons in `SourcePanel` similar to the one shown above for handling a click event on the radio button `file` above. You will need to prefix any variable which you refer to inside the body of the event handler with the keyword `final`, or promote the variable to be a private field of the class.

28. Add suitable import statements to `SourcePanel` so that `SourcePanel.java` compiles.

In order to use your new version of `SourcePanel` you will need to provide an implementation of each of the four abstract methods. Below is the code you should use to create a new instance of `SourcePanel` inside the method `createSourcePanel` found in `GuiLife`. Make sure you understand how it works and are prepared to answer questions on it from your Assessor.

```
JPanel result = new SourcePanel(){
 protected boolean setSourceFile() {
  JFileChooser chooser = new JFileChooser();
  int returnVal = chooser.showOpenDialog(this);
  if (returnVal == JFileChooser.APPROVE_OPTION) {
   File f = chooser.getSelectedFile();
   try {
    List<Pattern> list = PatternLoader.load(new FileReader(f));
    patternPanel.setPatterns(list);
    resetWorld();
    return true;
   } catch (IOException ioe) {}
  }
  return false;
 }
```

```
 protected boolean setSourceNone() {
  world = null;
  patternPanel.setPatterns(null);
  resetWorld();
  return true;
 }
 protected boolean setSourceLibrary() {
  String u = "http://www.cl.cam.ac.uk/teaching/current/ProgJava/nextlife.txt";
  return setSourceWeb(u);
 }
 protected boolean setSourceThreeStar() {
  String u = "http://www.cl.cam.ac.uk/teaching/current/ProgJava/competition.txt"
  return setSourceWeb(u);
 }
 private boolean setSourceWeb(String url) {
  try {
   List<Pattern> list = PatternLoader.loadFromURL(url);
   patternPanel.setPatterns(list);
   resetWorld();
   return true;
  } catch (IOException ioe) {}
  return false;
 }
};
```

29. Update your copy of `createSourcePanel` in `GuiLife` to create an instance of type `SourcePanel` using the definition of the anonymous inner class shown above.

30. Add the following import statements to `GuiLife`:

```
import javax.swing.JFileChooser;
import java.io.File;
import java.io.FileReader;
```

31. Compile and run your program. You should now be able to select between the four input sources and display the first pattern found in each source, except for "none" which should display no input sources in the Pattern Panel.

Selecting a pattern

The final piece of functionality missing so far is the ability to select a pattern using the Pattern Panel. Now that you have completed the rest of the code, this should be relatively easy given the following example event handler:

```
guiList.addListSelectionListener(new ListSelectionListener() {
 public void valueChanged(ListSelectionEvent e) {
  if (!e.getValueIsAdjusting() && (patternList != null)) {
   int sel = guiList.getSelectedIndex();
   if (sel != -1) {
    currentPattern = patternList.get(sel);
    onPatternChange();
   }
  }
 }
});
```

32. Create a new field called `patternList` in the class `PatternPanel`. Change the method `setPatterns` in the class `PatternPanel` to update the field `patternList` to reference the method argument `list` whenever the `setPatterns` method is called. (This will allow the event handler you write in the next step to access the list of patterns by referencing the field `patternList`.)

33. Add the event handler shown above to the constructor of `PatternPanel`.

34. Make the class `PatternPanel` an abstract class and add an abstract method called `onPatternChange` which should take no arguments and return no result.

35. Modify the declaration of `patternPanel` inside the method `createPatternPanel` in the class `GuiLife` to create an anonymous inner class which provides an implementation of `onPatternChange`. Your implementation of `onPatternChange` should do only one thing: call the `resetWorld` method. (This is sufficient since the `resetWorld` method calls `getCurrentPattern` to determine the current pattern and loads this pattern into the Game Panel.)

36. Test that your program works by loading a variety of patterns into it from disk and from the on-line library.

You may have noticed that changing the storage type used in the `ControlPanel` does not take effect until another pattern is selected in Pattern Panel or a new source is selected in the `Source Panel`. This behaviour is acceptable in this course, but a more polished implementation would solve this difficulty.

It's very slow

Try loading the '1A Java Ticker' pattern from the library. This is a pattern with the emergent behaviour that it displays a message across the screen. You will find that you have to be very patient waiting for this message to appear.

HashLife is an algorithm invented by William Gosper which provides significantly faster execution than our simple algorithm. The algorithm itself is complicated and beyond the scope of this course but an implementation is provided for you which you can download from http://www.cl.cam.ac.uk/teaching/current/ProgJava/hashlife.jar. The documentation is available online at http://www.cl.cam.ac.uk/teaching/current/ProgJava/hashlife. However, to make use of this code you only need to know that the jar file contains a class `HashWorld` inside the package `uk.ac.cam.acr31.life.hash` and that this class implements the `World` interface.

37. Download the jar file above and integrate `HashWorld` into your program. You should provide an additional radio button in the Control Panel to allow the user to select `HashWorld`.

This is an example of how important the choice of algorithm is to the execution speed of a program. The `ArrayLife` implementation will be faster than HashLife (and certainly consume less memory) on small worlds. On large worlds (particularly sparse ones), HashLife is significantly faster.

Tick 7

To complete your tick you need to prepare a jar file with the contents of all the classes you have written in this workbook and email it to `ticks1a-java@cl.cam.ac.uk`. Your jar file should contain:

```
uk/ac/cam/your-crsid/tick7/AgingWorld.java
uk/ac/cam/your-crsid/tick7/ArrayWorld.java
uk/ac/cam/your-crsid/tick7/ControlPanel.java
uk/ac/cam/your-crsid/tick7/GamePanel.java
uk/ac/cam/your-crsid/tick7/GuiLife.java
uk/ac/cam/your-crsid/tick7/HelloActionWorld.java
uk/ac/cam/your-crsid/tick7/PackedLong.java
uk/ac/cam/your-crsid/tick7/PackedWorld.java
uk/ac/cam/your-crsid/tick7/PatternFormatException.java
uk/ac/cam/your-crsid/tick7/Pattern.java
uk/ac/cam/your-crsid/tick7/PatternLoader.java
uk/ac/cam/your-crsid/tick7/PatternPanel.java
uk/ac/cam/your-crsid/tick7/SourcePanel.java
uk/ac/cam/your-crsid/tick7/Strings.java
uk/ac/cam/your-crsid/tick7/WorldImpl.java
```

together with *all* the class files generated when you compile these source files. Do not include any of the contents of `world.jar` or `hashlife.jar` in your submission. The submission system will continue to run throughout the vacation period, although our response to email queries might be a bit slower.

This concludes the Programming in Java course. Please make sure you fill out a feedback form for the course and give it to your ticker. The course lecturers are particularly interested in whether the practical class format has worked for you and whether the time and effort invested was worthwhile.