Appendix A

Event Driven Programming with a Graphical User Interface (GUI)

A.1 A Few Graphical Components

This appendix is meant to help you build event-driven programs with a graphical user interface (GUI). These programs are characterized by **graphical components**, such as windows, buttons, and menus that react to events generated by a user interacting with the graphical components through the mouse and keyboard. When building event-driven applications with a GUI, there are two things you have to do:

1. Make the graphical components visible to the user.
2. Make sure the correct things happen when an event occurs, such as the user clicking on something, moving a slider bar, selecting from a pull-down list, or pressing a key.

A graphical component is an object with a graphical appearance that can generate event objects when they are touched by users. Java has graphical components, such as buttons, labels, text fields, lists, slider bars, and a window to hold all of these graphical components. This section shows how to get a few graphical components into a window and make them visible to the user. In the next chapter, you will learn how to have the correct code respond to events generated by user interaction with the application.

This section uses a few of the many components in the `javax.swing` package. These classes help programmers build graphical user interfaces (GUIs). The first example will show graphical components constructed from the following `javax.swing` classes (or Swing classes):

- `JFrame`: A window with a title, border, submenu, and some buttons to the upper right.
- `JButton`: A graphical component that can "pushed" by clicking on the button.
- `JLabel`: A display area for a small amount of text.
- `JTextField`: A component that allows input and editing of a single line of text.

The program could import all four classes individually like this:

```
import javax.swing.JFrame;
import javax.swing.JButton;
import javax.swing.JLabel;
import javax.swing.JTextField;
```

However, since all of these classes are in the `javax.swing` package, you could simply use the following shortcut to gain access to the 90 or so classes in the package:

```
import javax.swing.*;  // Use any Swing component without qualifying the classes
```

The first object that will be constructed here is a window that holds other components. Java's `JFrame` class provides the means to contain other components. Some graphical components can contain other components, such as `JFrame` and `JPanel`. The `JFrame` is the base of the desktop application. will be constructed with a string that is the title of the window. To see the `JFrame`, you must send the `JFrame` object a `setVisible(true)` message. But before you do this, it is recommended that you send a message to the JFrame so the program will terminate when the user closes the window later on. To get these and more methods of `JFrame` simply extend it.

```
import javax.swing.*;  // Use any Swing component without qualifying the classes

// FirstGUI inherits all the methods of JFrame (setSize, add, setLayout...)
public class FirstGUI extends JFrame {

  public static void main (String[] args) {

    // Construct a window with its title and show it (it will be very small)
    JFrame theWindow = new FirstGUI();
  }

  public FirstGUI() {
    // Set the title of the window
    this.setTitle("Graffiti");

    // When the user closes the window, the program will terminate
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    // Make the window visible
    this.setVisible(true);
  }
}
```

Output

The window will be shown in the upper-left corner with a very small height and width. To make the window appear bigger, send a `setSize` message to the `JFrame` object. The first argument is the width—an integer representing the number of pixels. The second argument is the height, which is also measured in **pixels**—the smallest dots on your computer screen.

```
// Make the window 220 pixels wide and 100 pixels high
this.setSize(220, 100);
```

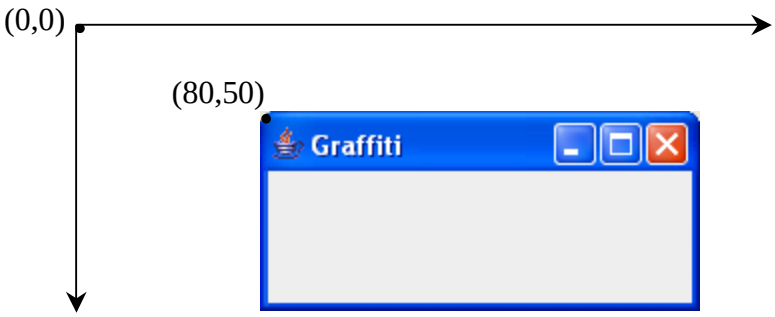
The following `setLocation` message will position the left side of the `JFrame` object 80 pixels to the right of the upper-left corner of the screen. The upper part of the `JFrame` will be shown 50 pixels below the top of the computer screen.

```
// Set the window's location from the upper-left corner of the screen
this.setLocation(80, 50);  // 80 pixels right, 50 pixels down
```

The coordinate system used to locate windows on a screen and to draw onto a surface uses the upper-left corner as position 0,0. Therefore, when this window is shown, the upper-left corner will be in position (80,50), 80 pixels to the right and 50 pixels down.

```
// Make the window visible
theWindow.setVisible(true);
```

Java's Coordinate System



Other graphical components can be added to a `JFrame` with the following two-step process:

1. Construct the components.
2. Add components to the `JFrame` (optionally referenced with `this`)

This example shows a button, a label, and a text field. All three objects are constructed with `String` arguments:

```
// Construct some components that will be added to the container
JButton clickMeButton = new JButton("Nobody is listening to me");
JLabel aLabel = new JLabel("Button above, text field below");
JTextField textEditor = new JTextField("You can edit this text");
```

At this point, `theWindow.setVisible(true)` would not show these three objects. You have to add them to `theWindow`. that can

```
// Add three components to the content pane part of the JFrame
// If this code is in the constructor, this. can be optionally used
this.add(clickMeButton, BorderLayout.NORTH);
add(aLabel, BorderLayout.CENTER);
add(textEditor, BorderLayout.SOUTH);
```

The first argument is the component you want to add to the window. The second argument describes the area to which you want to add the component. Now, a show message will show the `JFrame` with three other graphical components added.

When adding to a `JFrame` object, you can specify where on the `JFrame` the components will be added. There are five choices: `BorderLayout.NORTH`, `BorderLayout.SOUTH`,

`BorderLayout.WEST`, `BorderLayout.EAST`, and `BorderLayout.CENTER`. This is illustrated in the following `JFrame` that has five `JButton` objects, constructed with strings such as `"North"` and `"South"`, added to the content pane of the `JFrame`:

This `JFrame` illustrates Java's `BorderLayout` layout manager. It is one of several of Java's layout managers. A **layout manager** makes the decision on how to arrange graphical components in a `JFrame`. `BorderLayout` is the default for `JFrame`, but this can be changed with a `setLayout` message. Now here is the code discussed above, in one complete program.

```
// Construct a window and add some other graphical components to it
import java.awt.BorderLayout;
import javax.swing.*; // Use any Swing component without qualifying the classes

public class FirstGUI extends JFrame {

  public static void main(String[] args) {
    // Construct a window with its title and show it (it will be very small)
    JFrame theWindow = new FirstGUI();
  }

  public FirstGUI() {
    // Set the title of the window
    this.setTitle("Graffiti");

    // When the user closes the window, the program will terminate
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    // Make the window 220 pixels wide and 100 pixels high
    this.setSize(220, 100);
    this.setLocation(80, 50); // 80 pixels right, 50 pixels down

    JButton clickMeButton = new JButton("Nobody is listening to me");
    JLabel aLabel = new JLabel("Button above, text field below");
    JTextField textEditor = new JTextField("You can edit this text");

    // Add three components to the content pane part of the JFrame
    // If this code is in the constructor, this. can be optionally used
    this.add(clickMeButton, BorderLayout.NORTH);
    add(aLabel, BorderLayout.CENTER);
    add(textEditor, BorderLayout.SOUTH);

    // Make the window visible

    this.setVisible(true);
  }
}
```

At this point, nothing will happen when the button is clicked. This appendix later shows how to make it so you can specify what happens when the user interacts with the graphical user interface.

Layout Managers—`BorderLayout` and `GridLayout`

The way that graphical components are laid out as they get added to the `JFrame` is controlled by the container's layout manager. By default, a `JFrame` uses the `BorderLayout` layout manager to add graphical components. If the programmer fails to specify one of five locations where graphical components can be added, those components will be added to the center of the `JFrame`, one on top of another. Therefore, it is wise to specify one of the five areas as a second argument to `add`. The default layout manager is an instance of `BorderLayout`, which is defined in the `java.awt` package. The Bank Teller window shown above was built with a layout manager that is an instance of `GridLayout`, which is also defined in the `java.awt` package.

The layout manager for a `Container` was changed by means of the `Container` `setLayout` message. The `setLayout` message requires an instance of a layout manager, such as `GridLayout`. (There are several other such layout managers implemented in Java, and you can even define your own.) A `GridLayout` object can be constructed for maximum programmer control by using these `int` arguments:

1. The number of rows (`5`, below)
2. The number of columns (`2`, below)
3. The horizontal gap specified as the number of pixels between columns (`8`, here)
4. The vertical gap specified as the number of pixels between rows (`16`, here)

```
GridLayout fiveByTwo = new GridLayout(5, 2, 8, 16);
```

The layout manager object named `fiveByTwo` will control how graphical components are laid out in the JFrame with this `setLayout` message to this:

```
this.setLayout(fiveByTwo);
```

Now graphical components will be added in a row-wise fashion, beginning in the upper-left corner of the `JFrame`. The numbers shown in Figure 3.6 represent the ordering of how 10 graphical components will be added with a 5-row by 2-column `GridLayout` layout manager in charge. The shaded areas represent where the components will go. The unshaded areas represent the horizontal and vertical gaps between those graphical components.

`GridLayout`'s order of added graphical components

1	**2**
3	**4**
5	**6**
7	**8**
9	**10**

Self-Check

A.1 Fill in the code to make `buttonWindow` look as shown (assume the correct imports exist). Let there be four pixels between the `JButton` objects.


```
setTitle("6 Buttons");
setSize(150, 110);
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
// -------------Begin Answer

    Need several messages here

// --------------End of Answer
add(new JButton("1"));
add(new JButton("2"));
add(new JButton("3"));
add(new JButton("4"));
add(new JButton("5"));
add(new JButton("6"));
setVisible(true);
```

A.2 Event-Driven Programming

Chapter 1 introduced the Input/Process/Output pattern in the context of keyboard input and console output. Most programming assignments so far follow this mode: The user is prompted to enter input, the program does some processing, and the results are shown as output. For example, here is the problem specification along with one sample dialogue from Programming Project 1D: Seconds

1D Seconds

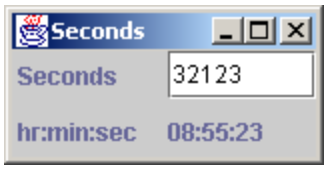
Write a program that reads a value in seconds and displays the number of hours, minutes, and seconds represented by the input. Your dialogue must look exactly like this:

```
Enter seconds: 32123
8:55:23
```

When prompted from a text-based console window, the user enters `32123` at the keyboard. Some actions occur behind the scenes. Then the result is displayed to the text-based console window.

Input/Process/Output with a Graphical Interface

The same Input/Process/Output pattern can be done in a graphical fashion. The prompt can be a `JLabel` object (**Seconds** in the example below). The keyboard is still used to enter input, but now the input goes into a `JTextField` object (with 32123 below). The output still goes to the screen, but this time in the form of a `JLabel` object (**08:55:23** below).



The user still enters input (32123), processing still occurs, and the result of the processing is still shown (**08:55:23**). It is the Input/Process/Output pattern with a different face, a graphical face, a graphical user interface. A **graphical user interface** (GUI) allows users to interact with programs through graphical components such as buttons, text fields, pull down lists, and menu selections.

Instead of users entering text in response to a prompt, this event-driven program will update the **hr:min:sec** fields whenever the user presses enter into a text field. Instead of a program that runs once with one user input, the user can enter many values to see many results. The program will not terminate until the user closes the window. This allows multiple Input/Process/Outputs.

Java's Event-Driven Programming Model

With Java, when a user enters text into a text field, that text field sends a message to another object. The text field does not perform the processing. Instead, the graphical component informs another object what has happened at the user interface: "Hey object, someone just entered input into my text field, do whatever it is you want to do—perform your actions."

The object that waits to be informed that a user has interacted with a particular graphical component is called a **listener**. The listener object does the processing and typically shows the result in the window. When a program has listener objects waiting for messages from graphical components in a interface, the program is called an **event-driven program** with a GUI.

An event-driven program needs classes built in a specific way. There is substantial overhead that requires some getting used to. However, after you have written a few event-driven programs with a graphical user interface, you can use the same approach for button clicks, menu selections, hyperlink events, mouse clicks, and any of the other events generated by Java graphical components, the mouse, and the keyboard.

Before looking at all of these details, first consider the complete event-driven GUI solution to Programming Project 1D Seconds. The numbers indicate the major differences from previous programs. The new concepts will be presented immediately after this complete example:

❶ A new import is needed for classes and interfaces related to events

❷ The class extends `JFrame`. This allows you to treat any new class as a `JFrame`

❸ An instance of the inner class that `implements ActionListener` is registered to listen to user input

❹ There is an inner class. This allows access to the needed graphical components

❺ The inner class implements the `ActionListener` interface so it can be registered as a listener

```
// Build a window with four components, two of which are needed by the listener
import javax.swing.*;           // JFrame, JTextField, JButton
import java.awt.GridLayout;     // GridLayout
import java.text.DecimalFormat; // To make "7" into "07"
❶
import java.awt.event.*;   // ActionListener, ActionEvent
                           ❷
public class TimeFrame extends JFrame  {
  public static void main(String[] args)   {
    JFrame window = new TimeFrame();
  }

  // The inner class methods can also reference these instance variables
  private JTextField secondsField;
  private JLabel hmsLabel;

  public TimeFrame()  {
    // These three messages are sent to the TimeFrame object being constructed
    setTitle("Seconds");
    setSize(160, 80);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    // Ask for the container of the TimeFrame object being constructed
    Container contentPane = getContentPane();
    contentPane.setLayout(new GridLayout(2, 2, 4, 4));

    // a) Construct the graphical component that will be listened to
    //    This will be needed by the inner class
    secondsField = new JTextField();

    // b) Add the component to the GUI with a prompt added first
    contentPane.add(new JLabel(" Seconds"));
    contentPane.add(secondsField);

    // c) Construct an instance of a class that has the actionPerformed method
    SecondsFieldListener inputListener = new SecondsFieldListener();

  ❸
    // d) Register the listener object to listen to the graphical component
    secondsField.addActionListener(inputListener);

    // Add the bottom row for output
    contentPane.add(new JLabel(" hr:min:sec"));
    hmsLabel = new JLabel("??:??:??"); // need this hmsLabel in inner class
    contentPane.add(hmsLabel);

    setVisible(true);
  }

  ❹
  ////////////////////// BEGIN INNER CLASS ////////////////////////////
  // An instance of this ActionListener will listen to a JTextField.
  // This inner class can reference secondsField and hmsLabel.
  // It also has its own DecimalFormat object to put in leading zeros.
  private class SecondsFieldListener implements ActionListener {  ❺
    private DecimalFormat formatter;

    public SecondsFieldListener()  {
      formatter = new DecimalFormat("00");
    }
```

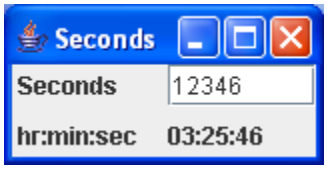
```
    public void actionPerformed(ActionEvent ae) { // Parameter not used
      // Get the seconds
      String intAsString = secondsField.getText();
      int total = Integer.parseInt(intAsString);

      // If an exception is thrown on bad input, this method terminates
      // Compute the hours, minutes, and seconds
      int hours = total / 3600;
      int minutes = total % 3600 / 60 ;
      int seconds = total % 60;

      // Format the output with leading zeros and colons
      String answer = formatter.format(hours) + ":" +
                      formatter.format(minutes) + ":" +
                      formatter.format(seconds);

      // Update the output label
      hmsLabel.setText(answer);
    }
  }
  ////////////////////// END INNER CLASS ////////////////////////////////

} // End class TimeFrame
```



Listeners

To get things done in an event driven GUI, an object must be enabled to execute the desired code whenever users of a GUI interact with those graphical components. In Java, when a user interacts with a graphical component, that object sends a message to the object that is "listening." The "listener" object then performs the actions that represent the appropriate response to the event.

In the case of a user clicking a button or entering text into a text field, the message is `actionPerformed`. Objects that have this `actionPerformed` method can be made to listen to buttons and text fields. Here is the method heading that is required by Java:

```
/** From the ActionListener interface
  *
  * The method that you must implement to get code to respond to button clicks
  * and user inp0ut into text fields.
  */
  public void actionPerformed(ActionEvent e);
```

Once everything is correct, `actionPerformed` messages will be sent to any object that was made to listen to a user interacting with a graphical component. To accomplish this, Java requires that you register objects to listen to graphical components. The method to do this for `JButton` and `JTextField` objects is named `addActionListener`. Here is documentation for the message that must be sent to the graphical component so it knows which objects are listening.

```
/** From the JButton and JTextField classes
  *
  * This JButton method registers an object to listen. aListener must be an
  * instance of a class that implements the ActionListener interface. After this
  * message registers the listener, actionPerformed messages can be sent so
  * there is a response to the user interaction.
  */
  public void addActionListener(ActionListener aListener)
```

The argument passed to the parameter `aListener` is the object that will receive an `actionPerformed` message. Therefore, this argument must be an `ActionListener` object. However, there is no `ActionListener` class. You must write one that is appropriate for your own application. Java's event model requires that listener objects implement the `ActionListener` interface.

Java does this to ensure that later on, when the graphical component sends an `actionPerformed` message to the listener, the `ActionListener` object will understand the `actionPerformed` message. This avoids runtime errors later on that are more difficult to detect. With this design—the `ActionListener` parameter required—you will get a compiletime error, which is easier to deal with. This is part of the overhead to get event-driven programming to work in Java. The objects that listen to buttons and text fields must be instances of a class that implements Java's `ActionListener` interface.

When implementing an interface, you need to add `implements` *SomeInterfaceName* to the class heading, retype the method heading(s) of the interface, and implement the method(s) in an appropriate manner. Before looking at the class that implements `ActionListener`, consider the steps required in the Seconds example so that one object can listen to a graphical component. Here is the code of the objects and messages required to register a listener object for the current example. The graphical component can later send `actionPerformed` messages to the correct object.

```
// a) Construct the graphical component that will be listened to
secondsField = new JTextField();    // this will be needed by the inner class

// b) Add the component to the GUI with a prompt added first
add(new JLabel(" Seconds"));
add(secondsField);

// c) Construct an instance of a class that has an actionPerformed method
SecondsFieldListener inputListener = new SecondsFieldListener();

// d) Register the listener object to listen to the text field
secondsField.addActionListener(inputListener);
```

With this code, when the user enters text into `secondsField`, the `secondsField` object will send an `actionPerformed` message to `inputListener`. This `actionPerformed` method must be in a class that implements `ActionListener`.

Have a Class Implement ActionListener

The one thing still missing in the inner class named `SecondsFieldListener`. An instance of `SecondsFieldListener` must be assigned to the `ActionListener` parameter of the `addActionListener` method. This means that the listener class must implement the `ActionListener` interface. This demand in turn demands that the class have the `actionPerformed` method. It is in this `actionPerformed` method where you must write the code you want to be executed whenever the button is clicked. The class begins like this:

```
class SecondsFieldListener implements ActionListener {
    // ...
}
```

With the Java keyword `implements` and the interface name `ActionListener`, this class will not compile until you implement the `actionPerformed` method. The `actionPerformed` method must have the same method heading as specified in Java's `ActionListener` interface. The next steps are to add the method specified in the interface and replace the semicolon (`;`) at the end with a pair of curly braces (an additional import is also needed for the `ActionListener` interface and the `ActionEvent` class).

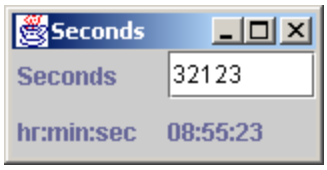
```
import java.awt.event.*;  // For interface ActionListener and class ActionEvent

class SecondsFieldListener implements ActionListener {

  public void actionPerformed(ActionEvent anActionEvent)  {
  // Write the code that should execute when the user interacts with the text field
  }
}
```

The code that is written between the curly braces in `actionPerformed` is the code that will execute each time the user enters text into the text field labeled with Seconds.[1] The GUI is repeated here for your convenience:

[1]Although the method heading requires an `ActionEvent` parameter, you are not required to use this parameter anywhere in your code.



Use an Inner Classes (it is convenient)

Because the code in actionPerformed will need access to the text field for input and the `JLabel` for output, the `ActionListener` class just started will be implemented as an inner class. An **inner class** is an entire class that is implemented inside of another class. An inner class has access to the private instance variables of the enclosing class. For example, with `SecondsFieldListener` implemented inside of the `TimeFrame` class, the `actionPerformed` method will have access to the `JTextField` to get the user input. This technique is typical of Java programs when a listener object needs access to several objects in a `JFrame`.[2] This inner class is `private` since no one else needs it.

```
private class SecondsFieldListener implements ActionListener   {

  private DecimalFormat formatter;

  public SecondsFieldListener() {
    formatter = new DecimalFormat("00");
  }

  public void actionPerformed(ActionEvent e)  {
    // TBA
  }
}
```

In this particular example, integer output will be formatted with leading zeros if necessary, so this inner class will have an instance of `DecimalFormat` to help it do its job. The constructor specifies that format messages will always show two digits.

The next step is to write the code that will execute each time the user enters input into the text field. However, before completing the required `actionPerformed` method, first consider three useful methods that will be needed for `actionPerformed` to accomplish its tasks.

[2] Java has anonymous inner classes (no class heading necessary) for this purpose, but they require more syntax. Anonymous inner classes are not covered in this textbook. Private inner classes are used instead.