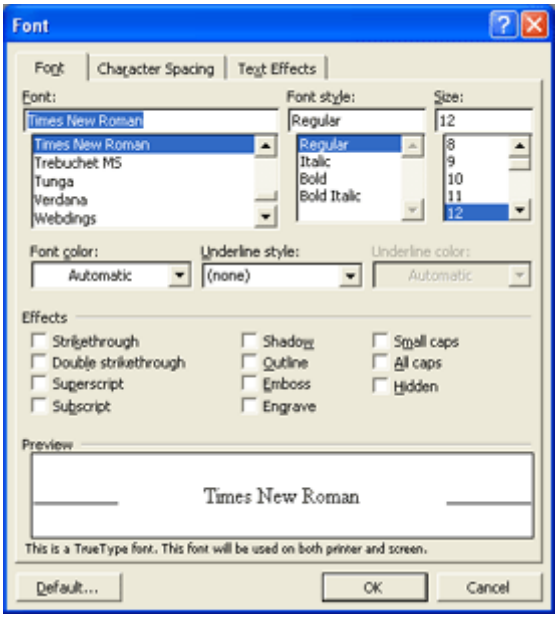
Chapter 6 - Java GUI Applications

Graphical User Interfaces (GUIs) are mechanisms for allowing users to enter data in the most economical and straightforward manner possible. Figure 6.1, "Example GUI: Font Dialog" shows an example GUI that is designed to allow a user to choose a font type, style and size. There are different controls such as buttons, lists, choice items and check box items. This dialog box allows the user to choose the options easily, while it also allows the programmer to carefully control the way that the user can enter the data, preventing the user from entering invalid options.

Figure 9.1. Example GUI: Font Dialog



Java provides two mechanisms for developing user interface applications in Java - AWT and Swing. AWT (Abstract Windowing Toolkit) is the mechanism we will use here, but we will discuss Swing later. The AWT is tied directly to the operating system (Figure 6.1, "Example GUI: Font Dialog" is on the Windows OS) and so AWT applications will have a different "look-and-feel" on different operating systems.

All source code examples for this chapter are available on Github:
https://github.com/derekmolloy/ee402/tree/master/notes_examples/chapter6

AWT Components

A component is an object with a graphical representation that can be displayed on the screen and that can interact with the user. The Component class is the abstract parent of the nonmenu-related AWT (Abstract Window Toolkit) components.

Button (java.awt.Button)

To create a Button object, simply create an instance of the Button class by calling one of the constructors. The most commonly used constructor of the Button class takes a String argument, that gives the Button object a text title. The two constructors are:

```
Button()              // Constructs a Button with no label.
Button(String label)  // Constructs a Button with the specified label.
```

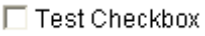
Figure 6.2. A Button Component

When a user presses on a Button object, an event is generated. The label of the button that is pressed can be obtained.

Checkboxes (java.awt.Checkbox)

Checkboxes have two states, on and off. The state of the button is returned as the Object argument, when a Checkbox event occurs. To find out the state of a checkbox object we can use getState() that returns a true or false value. We can also get the label of the checkbox using getLabel() that returns a String object.

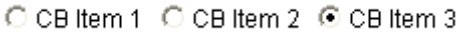
Figure 6.3. A Checkbox Component



Radio Buttons (java.awt.CheckboxGroup)

Is a group of checkboxes, where only one of the items in the group can be selected at any one time.

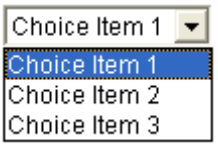
Figure 6.4. A Radio Button Component



Choice Buttons (java.awt.Choice)

Like a radio button, where we make a selection, however it requires less space and allows us to add items to the menu dynamically using the addItem() method.

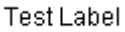
Figure 6.5. A Choice Button Component



Labels (java.awt.Label)

Allow us to add a text description to a point on the applet or application.

Figure 6.6. A Label Component



TextFields (java.awt.TextField)

Are areas where the user can enter text. They are useful for displaying and receiving text messages. We can make this textfield read-only or editable. We can use the setEditable(false) to set a textfield read-only. There are numerous ways that we can construct a Textfield object:

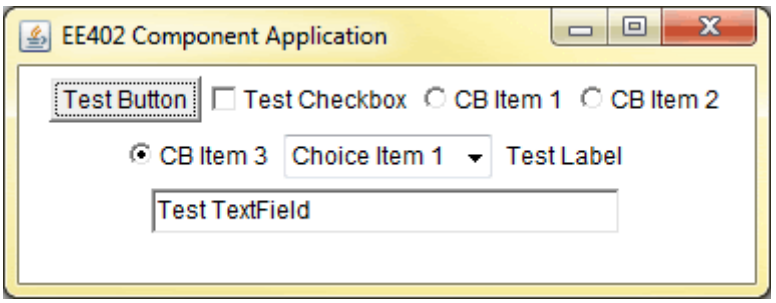
```
TextField text1 = new TextField();                // no properties
TextField text2 = new TextField("Some text");     // a textfield with a
                                                  // predefined String
TextField text3 = new TextField(40);              // a textfield with a
                                                  // predefined size
TextField text4 = new TextField("Some text", 50); // combination of the two
```

Figure 6.7. A TextField Component

An Example Component Application

To show these components in action we can write a short piece of code that displays these components, but they do not have any events behind them - they work visually but do not have any true program function. Figure 6.8, "A Component Application" shows an example application that details all the previous components.

Figure 6.8. A Component Application



You can see this application running in Figure 6.9.

ComponentApplication.java

```
package ee402;

import java.awt.*;

@SuppressWarnings("serial")
public class ComponentApplication extends Frame {

        public ComponentApplication() {
                super("EE402 Component Application");
                this.setLayout(new FlowLayout());

                Button b = new Button("Test Button");
                this.add(b);

                Checkbox cb = new Checkbox("Test Checkbox");
                this.add(cb);

                CheckboxGroup cbg = new CheckboxGroup();
                this.add(new Checkbox("CB Item 1", cbg, false));
                this.add(new Checkbox("CB Item 2", cbg, false));
                this.add(new Checkbox("CB Item 3", cbg, true));

                Choice choice = new Choice();
                choice.addItem("Choice Item 1");
                choice.addItem("Choice Item 2");
                choice.addItem("Choice Item 3");
                this.add(choice);

                Label l = new Label("Test Label");
                this.add(l);

                TextField t = new TextField("Test TextField",30);
                this.add(t);

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

        public static void main(String[] args) {
                new ComponentApplication();
         }
}
```

Layout Policies

If we want to position textboxes, buttons and checkboxes on a window like in Figure 6.1, "Example GUI: Font Dialog", we need a way to organize these components in the window. A layout policy allows us a way to describe the order that we want to display

these components, allowing a structure for placing these components on the application. In the last example we used a FlowLayout, as it is a very easy layout manager, that just adds the components to the application in the order in which you use the add() method.

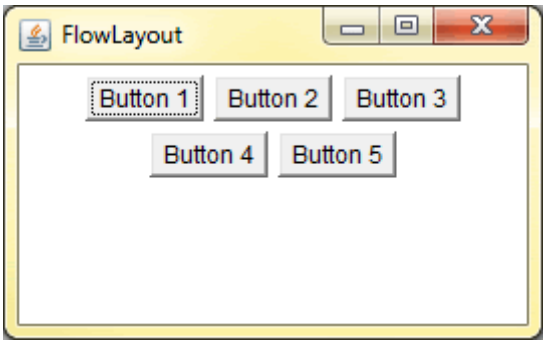
It is possible to change the layout of the components using a layout manager, the most common layouts are:

- FlowLayout
- GridLayout
- BorderLayout

FlowLayout (java.awt.FlowLayout)

A FlowLayout arranges components left-to-right top-to-bottom, much like the "centred text" button in Microsoft Word for Windows, where each line is filled and centered until the line is full. In a flow layout form each component takes its preferred size.

Figure 6.9. A Flow Layout Example



FlowLayoutApp.java

```java
package ee402;

import java.awt.*;

@SuppressWarnings("serial")
public class FlowLayoutApp extends Frame {

	public FlowLayoutApp() {
		super("FlowLayout");
		this.setLayout(new FlowLayout());

		Button button1 = new Button("Button 1");
		Button button2 = new Button("Button 2");
		Button button3 = new Button("Button 3");
		Button button4 = new Button("Button 4");
		Button button5 = new Button("Button 5");
		add(button1);
		add(button2);
		add(button3);
		add(button4);
		add(button5);

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

	public static void main(String[] args) {
		new FlowLayoutApp();
	}
}
```

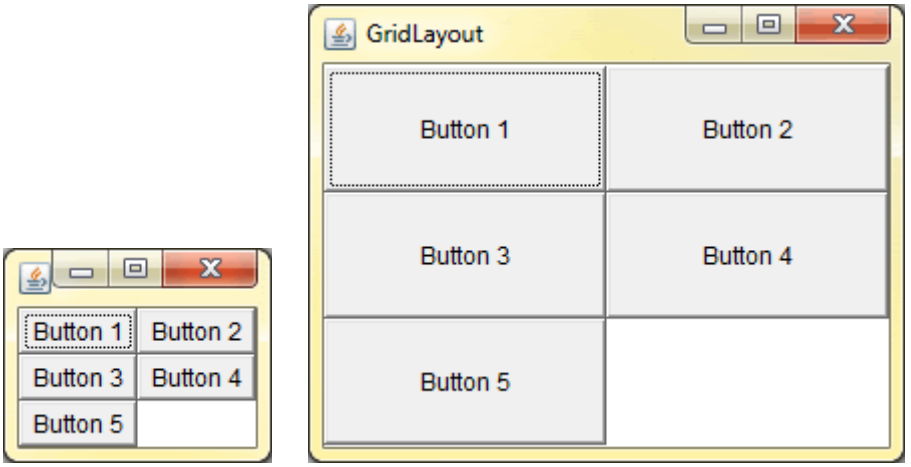
GridLayout (java.awt.GridLayout)

A GridLayout lays out the components in a rectangular grid. The container that you apply the layout to, divides the area into equal sized squares and each component that you add is sized to fit that square. If you look at Figure 6.10, "A Grid Layout Example" you will notice that the buttons have been scaled larger to fit the grid area exactly in (a). This can be difficult to get used to as when you scale the window as in (b) the buttons will resize to fill the space.

You must specify the number of rows and columns in the grid. So to construct a 3x2 grid as in Figure 9.10, "A Grid Layout Example" I used new GridLayout(0,2) to construct the nx2 GridLayout object. If you place a 0 for either the row or the column value to the constructor then you will have unlimited rows or columns. For example, new GridLayout(0,2) will construct a GridLayout object with 2 columns, but an unlimited number of rows.

The order that you use the add() method is still important as it adds the components to the grid from left-to-right top-to-bottom.

Figure 6.10. A Grid Layout Example



(a) preferred size (b) dragged and scaled

GridLayoutApp.java

```java
package ee402;

import java.awt.*;

@SuppressWarnings("serial")
public class GridLayoutApp extends Frame {

	public GridLayoutApp() {
		super("GridLayout");
		this.setLayout(new GridLayout(0,2));

		Button button1 = new Button("Button 1");
		Button button2 = new Button("Button 2");
		Button button3 = new Button("Button 3");
		Button button4 = new Button("Button 4");
		Button button5 = new Button("Button 5");
		add(button1);
		add(button2);
		add(button3);
		add(button4);
		add(button5);

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

	public static void main(String[] args) {
		new GridLayoutApp();
	}
}
```

BorderLayout (java.awt.BorderLayout)

A BorderLayout arranges a container into five regions, North, South, East, West and Center. Each region can have either one or zero components. If you do not place a component in the region then it will not be displayed. When we are adding components to a container we could write something like:

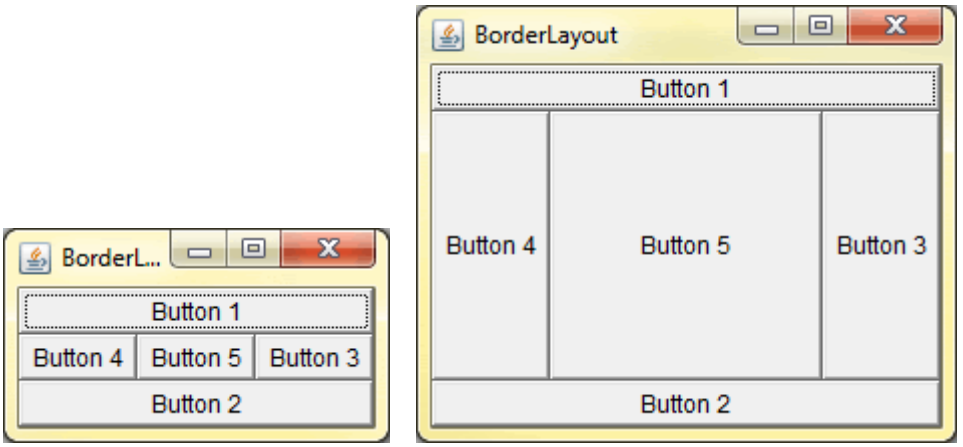
```java
Button testButton = new Button("Test Button");
this.add("North", testButton);

//or using add in another way

Button testButton = new Button("Test Button");
this.add(testButton, BorderLayout.NORTH);
```

When you add a component to the BorderLayout the components are added according to the components preferred size and the BorderLayout's preferred size. For Example as in Figure 6.11, "A Border Layout Application", a button added to the North will have the preferred height of the Button object, but it will have the preferred width of the BorderLayout container area. If you look at Figure 6.11, "A Border Layout Application" you will also notice that the "Button 3" object has its preferred width (in both (a) and (b)) of the Button object, but it has the preferred height of the "West" region of the BorderLayout container area.

Figure 6.11. A Border Layout Application



(a) The preferred Application Size (b) When the Application is Scaled

BorderLayoutApp.java

```java
package ee402;
import java.awt.*;

@SuppressWarnings("serial")
public class BorderLayoutApp extends Frame {
	public BorderLayoutApp() {
		super("BorderLayout");
		//this.setLayout(new BorderLayout());  default layout for a Frame

		Button button1 = new Button("Button 1");
		Button button2 = new Button("Button 2");
		Button button3 = new Button("Button 3");
		Button button4 = new Button("Button 4");
		Button button5 = new Button("Button 5");
		add(button1, BorderLayout.NORTH);
		add(button2, BorderLayout.SOUTH);
		add(button3, BorderLayout.EAST);
		add(button4, BorderLayout.WEST);
		add(button5, BorderLayout.CENTER);

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

	public static void main(String[] args) {
		new BorderLayoutApp();
	}
}
```

There is one other Layout Manager that is commonly used - the GridBagLayout. It allows components to be aligned vertically and horizontally without requiring the components to be the same size. You can see more information on this Layout Manager in the Java API documentation under java.awt.GridBagLayout.

The Panel Container (java.awt.Panel)

A Panel is the simplest form of container. A container is a component that can contain further components, or even other panels. The default layout for a Panel is the FlowLayout. Using Panel components allow us to develop quite advanced layouts as we can embed layouts within layouts. You can create a Panel object using:

```java
Panel p = new Panel();           //for a simple panel

//or

Panel P = new Panel(new BorderLayout()); //change the layout
```

More AWT GUI Components

Java provides another set of components that are useful in developing GUI applications.

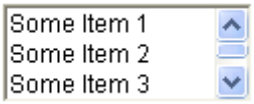
List (java.awt.List)

List components objects are useful for holding a large number of data items. They are quite similar to the Choice component, however they can allow for the multiple selection of items. The list automatically adds a scrollbar if it is required. The list constructor takes two arguments, the int number of items to be displayed and a boolean to allow/disallow multiple item selection.

```
3   List aList = new List(3, true);
4   aList.addItem("Some Item 1");
5   aList.addItem("Some Item 2");
6   aList.addItem("Some Item 3");
7   aList.addItem("Some Item 4");
8   //etc.
```

Clicking on an item selects it and pressing again on the same item deselects it. The selection works the same way as Windows explorer file selection. i.e., if you hold the SHIFT key and select with the mouse you can select from one item in The getSelectedItem() (or getSelectedItems() for multiple selections) method can be used to retrieve the currently selected item.

Figure 6.13. A List Component



TextArea (java.awt.TextArea)

Are used for creating areas of screen input where we want to provide large amounts of text data, either for displaying a text file, or implementing a text editor. TextArea components do not generate events. They can be set read-only or editable by using the setEditable() method.

Figure 6.14. A TextArea Component

The following are the constructors for a TextArea component

```
TextArea()                          // empty string as text.
TextArea(int rows, int columns)     // number of rows and columns specified
TextArea(String text)               // with the specified text.
TextArea(String text, int rows, int columns)
        // with the specified text, and with the specified number of
        // rows and columns.
TextArea(String text, int rows, int columns, int scrollbars)
        // with the specified text, and with the rows, columns, and
        // scroll bar visibility as specified either as SCROLLBARS_BOTH,
        // SCROLLBARS_HORIZONTAL_ONLY, SCROLLBARS_NONE
        // or SCROLLBARS_VERTICAL_ONLY
```

The TextArea object can be created as follows:

```
TextArea t = new TextArea("Test TextArea", 30, 5,
                          TextArea.SCROLLBARS_VERTICAL_ONLY);
```

Scrollbar (java.awt.Scrollbar)

Scrollbar objects are used to develop sliders that choose values in a certain range of data. We can use a Scrollbar to control an input value, scroll an image, and numerous other applications. The constructor of a Scrollbar takes five parameters:

- The orientation of the scrollbar, HORIZONTAL or VERTICAL.
- The initial value of the slider.
- The thickness of the slider drag component.
- The minimum value of the scrollbar.
- The maximum value of the scrollbar.

Use the getValue() method to read the current value of the Scrollbar object. Use the setValue() method to set the slider position in the scrollbar.

Figure 6.15. A Scrollbar Component

So an example use could be:

```
Scrollbar sb = new Scrollbar(Scrollbar.HORIZONTAL, 50, 10, 0, 100);
```

Component Events

Introduction

At the moment we have displayed components, adding them to applications, and they work to the extent that you can press the buttons, select from the list etc., however, they do not currently interact with your own code. If we want to use these components we must be aware of how the java.awt.event package classes work. Note that with

events the structure is quite different from Java 1.1.x (Java 1) to Java 1.2+.x (Java 2). There are many reasons for this, but the main reason is that there were serious problems with the event structure of Java 1, especially in relation to threaded applications. We are only examining the event structure of Java 2+.

ActionEvent (java.awt.event.ActionEvent)

An ActionEvent object is generated by several of the components we have just discussed. A component such as a Button object generates an ActionEvent object when it is pressed. We also need to register a listener with the component so that we can direct the event to do something in our application. For a Button object we can do this using, for example:

```
Button b = new Button("Test Button");
b.addActionListener(this);
```

Where this refers to an object that is capable of handling the ActionEvent object that is passed to it. So in this case the class in which the Button b is defined must implement that ActionListener interface. So for a full example, use the following application:

Figure 6.16. Button Events Application

The application should display which button is pressed in the TextField as soon as the button is pressed. The source code for this example is below.

ButtonEvents.java

```
1  package ee402;
2
3  import java.awt.*;
4  import java.awt.event.*;
5
6  @SuppressWarnings("serial")
7  public class ButtonEvents extends Frame implements ActionListener{
8
9                  private Button button1, button2, button3;
10                 private TextField status;
11
12                 public ButtonEvents()
13                 {
14                 super("Button Events");
15                 this.setLayout(new FlowLayout());
16                 status = new TextField(20);
17
18                 this.button1 = new Button("Button 1");
19                 this.button2 = new Button("Button 2");
20                 this.button3 = new Button("Button 3");
21
22                 this.button1.addActionListener(this);
23                 this.button2.addActionListener(this);
24                 this.button3.addActionListener(this);
25
26                 this.add(status);
27                 this.add(button1);
28                 this.add(button2);
29                 this.add(button3);
30
31                 this.pack();
32                 this.setVisible(true);
33                 }
34
35                 public void actionPerformed(ActionEvent e)
36                 {
37                 if (e.getActionCommand().equals("Button 1"))
38                 {
39                   status.setText("Button 1 Pressed");
40                 }
41                 else if (e.getActionCommand().equals("Button 2"))
42                 {
43                   status.setText("Button 2 Pressed");
44                 }
45                 else
46                 {
47                   status.setText("Button 3 Pressed");
                   }
```

```
48              }

                public static void main(String[] args) {
                        new ButtonEvents();
                }
}
```

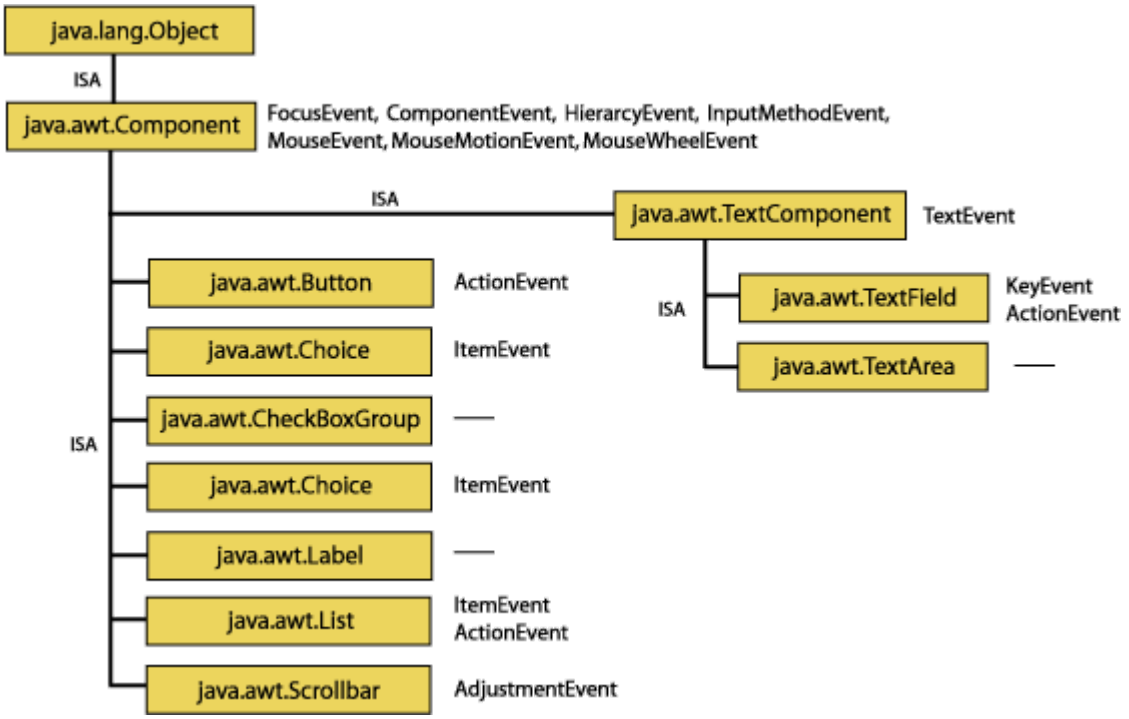
Some notes about this code segment:

- The java.awt.event package must be included to allow for the use of the ActionEvent and associated classes.
- The application class must implement ActionListener to handle ActionEvents. Once our class uses the statement implements ActionListener then we must write a method public void actionPerformed(ActionEvent e). If we do not write this method then the code will not compile.
- We must use the method addActionListener() to the component to state that this component generates events and to direct these events to a suitable object. In this application we use the keyword this which represents this object of the application. So we are sending the event to this object. We are **only** allowed to do that because this object implements ActionListener. If it did not then an error would be generated at compile time.
- Once the event is sent to this object then the actionPerformed() method is called and it is passed an ActionEvent object - in this case called e. We can then interrogate this ActionEvent object to determine its source. In this application we have three different buttons that could have caused this object to be generated. One way of interrogating the ActionEvent object is to use its getActionCommand() method that returns the action command as a String object. In the API documentation it states that the action command for a Button object is the text on the surface of the button by default. We then use the equals() method of the String class that allows us to compare the action command to a String such as "Button 1". If the "Button 1" String is a perfect match then we set the text of the TextField to be "Button 1 Pressed" using the setText() method of the TextField object status.

Other Event Types

The ActionEvent class and the ActionListener interface mechanism can be used to deal with events on buttons and on other components also. The ActionEvent class deals with actions that can occur. Other events can be created by other components - how do we know which events are created by which components? Well you have to read the API documentation for that component. For the components we have looked at in this section:

Figure 6.17. Component Events



- Button objects generate an ActionEvent object when the mouse is pressed on the Button object. You can add a listener to the object by using the addActionListener()method that requires a listener object as a parameter. The Listener object must implement the ActionListener interface that requires an actionPerformed() method to be written.
- Checkbox objects generates an ItemEvent object. You can add a listener to the object using the addItemListener() method that requires a listener object as a parameter. The listener must implement the ItemListener interface and is therefore required to write an itemStateChanged() method.
- A CheckboxGroup object is really a group of Checkbox objects and does not generate any events as a group, rather the individual objects generate the events as for Checkbox objects. If you need to find out the individual Checkbox object that was chosen use the getSelectedCheckbox() method.
- Choice objects generates an ItemEvent object. You can add a listener to the object using the addItemListener() method that requires a listener object as a parameter. The listener must implement the ItemListener interface and is therefore required to write an itemStateChanged() method.
- Label objects do not generate events.
- TextField objects generate KeyEvent objects when a key is pressed in the box. This KeyEvent object can be a key pressed, a key released or a key typed. You can use the addKeyListener() to add a listener to the TextField object. This method requires an object that has implemented the KeyListener interface that requires the object to implement the keyPressed(), keyReleased() and keyTyped() methods. A TextField object can also generate an ActionEvent object when the enter key is pressed in the text area. This ActionEvent has the same format as for the Button component above.
- List objects generate an ItemEvent object when items are selected or deselected in the list. You can add a listener to the object using the

addItemListener() method that requires a listener object as a parameter. The listener must implement the ItemListener interface and is therefore required to write an itemStateChanged() method. The Listobject also generates an ActionEvent object if the mouse is double-clicked or Enter is pressed on an item in the list.
- TextArea objects do not generate events.
- Scrollbar objects generate an AdjustmentEvent object when the Scrollbar's bubble is moved. You can add a listener to the Scrollbar object by using the addAdjustmentListener() method that requires an object that implements the AdjustmentListener interface requiring the adjustmentValueChanged() method to be written.

The Component class can generate many event types. The Component is the parent of all visible AWT components, providing the template for all AWT components. It can have:

- A FocusEvent which notifies a FocusListener listener when the component has been selected.
- A ComponentEvent which notifies a ComponentListener listener when the component has been resized, moved or made visible/invisible.
- A HierarchyEvent which notifies a HierarchyListener when the hierarchy that this component belongs to changes. A component can be added to a container's hierarchy using the add() method as we have used for myPanel.add(new Label("test")); where myPanel is an object of the Panel class, thus is a Container object.
- An InputMethodEvent which notifies a ComponentListener listener when the component has been re-sized, moved or made visible/invisible.
- A MouseEvent which notifies a MouseListener listener that the mouse has entered or exited the component, clicked or pressed/released the mouse button on the Component object.
- A MouseMotionEvent which notifies a MouseMotionListener listener when the mouse has been dragged or moved in the Component object area.
- A MouseWheelEvent which notifies a MouseWheelListener listener when the mouse wheel is rotated within the Component object area. The listeners mouseWheelMoved() method is called when the mouse wheel event occurs.

The TextComponent class can generate TextEvent objects. It has the Component class as its parent and so can generate the same events as above for Component. A TextEvent is generated when the object's text is changed.

Using Other Components

Introduction

Now that we have seen some of the other components that are available and we have also seen quite a complex event structure for these components an example should help clear up any confusion.

A Scrollbar Example

Task: Write the application as shown in Figure 6.18, "A Scrollbar Example" where the Scrollbar component updates the text in the TextField component when it is moved. The Reset Button component should set the value of the TextField object to 0. The Scrollbar component should have the range 0 to 100.

Figure 6.18. A Scrollbar Example

Solution: You can see this Scrollbar example running in Figure 6.18. The code is at the bottom of the page.

ScrollbarExample.java

```java
package ee402;

import java.awt.*;
import java.awt.event.*;

@SuppressWarnings("serial")
public class ScrollbarExample extends Frame
           implements AdjustmentListener, ActionListener {
        private Scrollbar scrollbar;
        private TextField status;
        private Button resetButton;

        public ScrollbarExample(){
               super("Scrollbar Example");
               this.setLayout(new GridLayout(2,1));
               Panel topPanel = new Panel();

               this.status = new TextField(20);
               this.scrollbar = new Scrollbar(Scrollbar.HORIZONTAL, 50, 10, 0, 110);
               this.resetButton = new Button("Reset");

               this.resetButton.addActionListener(this);
               this.scrollbar.addAdjustmentListener(this);

               topPanel.add(status);
               topPanel.add(resetButton);

               this.add(topPanel);
               this.add(scrollbar);

               this.updateText();
               this.pack();
               this.setVisible(true);
        }

        public void actionPerformed(ActionEvent e){
                if (e.getActionCommand().equals("Reset")){
                        this.scrollbar.setValue(0);
                        this.updateText();
                }
        }

        public void adjustmentValueChanged(AdjustmentEvent e){
                if (e.getSource().equals(scrollbar)){
                        this.updateText();
                }
        }
```

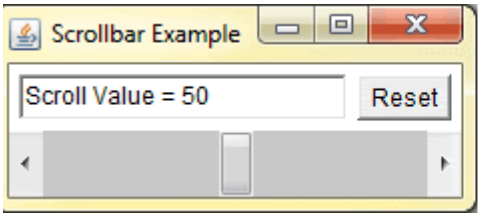
```
48          private void updateText(){
49                  status.setText("Scroll Value = " + scrollbar.getValue());
50          }
51
52          public static void main(String[] args) {
53                  new ScrollbarExample();
54          }
55  }
56
```

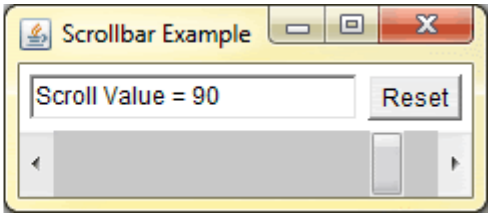
To complete this application there are a few points to note:

- The ScrollbarExample class implements both the AdjustmentListener and the ActionListener interfaces. A class can implement as many interfaces as desired. To implement both these interfaces an adjustmentValueChanged() method and an actionPerformed() method both had to be written, otherwise the code would not compile.
- I wrote a private method updateText() that updates the status state. I did this as otherwise there would be three different status.setText() calls, replicating code and leading to unpredictable results if two were changed and one was not.
- For the layout I used a 2x1 (i.e. 2 rows, 1 column) GridLayout with a panel (topPanel) in the top grid section and the scrollbar object in the bottom grid section. To the topPanel container I added the status TextField object and the resetButton Button object. Since the default layout for a Panel object is FlowLayout the components added are given their preferred size. Note that because the scrollbar object was added directly to the grid section it is bound to the grid size and will change size as the application changes size.
- I have called this.updateText() in the constructor so that the status TextField component will display the initial value of the scrollbar object.
- Note that in the constructor call to Scrollbar the range was set from 0 to 110. We were required by the task to allow a value to be chosen from 0 to 100. We set the maximum at 110 as the width of the bubble is 10 and the value selected is taken from the start of the bubble. So the maximum range value is always the value desired plus the width of the bubble. Note that when you run this application (alternatively see Figure 6.18, "A Scrollbar Example Application") you can press on the "arrows" at the left and right of the Scrollbar component and the value will increment/decrement by 1. If you press on the area between the bubble and the arrows then the value will increment/decrement by 10 - the width of the bubble.

Adding Window Events

At this point we have our application working, but what if we would like the application to quit when we press the 'X' in the top right-hand corner of the window? Well we need to look at Window Events.





If we look at the class Frame in the Java API documentation we can see that it has multiple parent classes. The API describes it as:

java.awt

Class Frame

- java.lang.Object
  - java.awt.Component
    - java.awt.Container
      - java.awt.Window
        - java.awt.Frame
- **All Implemented Interfaces:**
  ImageObserver, MenuContainer, Serializable, Accessible
  Direct Known Subclasses:
  JFrame

This means that any method that is available in the Window class or the Container class etc. is also present in the Frame class. If we read more in the API documentation we can see that:

Frames are capable of generating the following types of WindowEvents:

- WINDOW_OPENED
- WINDOW_CLOSING:
  If the program doesn't explicitly hide or dispose of the window while processing this event, the window close operation is canceled.
- WINDOW_CLOSED
- WINDOW_ICONIFIED
- WINDOW_DEICONIFIED
- WINDOW_ACTIVATED
- WINDOW_DEACTIVATED
- WINDOW_GAINED_FOCUS
- WINDOW_LOST_FOCUS
- WINDOW_STATE_CHANGED

So, the events are associated with the Window class (as they are WindowEvents) and if we go down we can see:

Since:

JDK1.0

See Also:

WindowEvent, Window.addWindowListener(java.awt.event.WindowListener), Serialized Form

That we should look at the addWindowListener() method, which expects an object that has implemented the WindowListener interfaces. This interface is described as having the following methods that you must implement:

Modifier and Type	Method and Description
void	**windowActivated(WindowEvent e)** Invoked when the Window is set to be the active Window.
void	**windowClosed(WindowEvent e)** Invoked when a window has been closed as the result of calling dispose on the window.
void	**windowClosing(WindowEvent e)** Invoked when the user attempts to close the window from the window's system menu.
void	**windowDeactivated(WindowEvent e)** Invoked when a Window is no longer the active Window.
void	**windowDeiconified(WindowEvent e)** Invoked when a window is changed from a minimized to a normal state.
void	**windowIconified(WindowEvent e)** Invoked when a window is changed from a normal to a minimized state.
void	**windowOpened(WindowEvent e)** Invoked the first time a window is made visible.

So, let us modify our previous code so that when the window activates that the scroll value is set back to 50 and when the 'X' in the top right-hand side is clicked that the program exits.

So, we want to implement the WindowListener interface in our class. Unfortunately, we have to implement all seven methods or our code will not compile. So, we will add custom code for windowClosing() to exit our application and custom code for windowActivated() to set the value to 50 every time you activate the window (e.g. maximising after minimising). The code looks like this:

ScrollbarExampleEvents.java

```
1  package ee402;
2
3  import java.awt.*;
4  import java.awt.event.*;
5
6  @SuppressWarnings("serial")
7  public class ScrollbarExampleEvents extends Frame implements AdjustmentListener,
8  ActionListener, WindowListener
9  {
10         private Scrollbar scrollbar;
11         private TextField status;
12         private Button resetButton;
13
14         public ScrollbarExampleEvents(){
15               super("Scrollbar Example");
16               this.setLayout(new GridLayout(2,1));
17               Panel topPanel = new Panel();
18
19               this.addWindowListener(this);
20
21               this.status = new TextField(20);
22               this.scrollbar = new Scrollbar(Scrollbar.HORIZONTAL, 50, 10, 0, 110);
23               this.resetButton = new Button("Reset");
24
25               this.resetButton.addActionListener(this);
26               this.scrollbar.addAdjustmentListener(this);
27
28               topPanel.add(status);
29               topPanel.add(resetButton);
30
31               this.add(topPanel);
32               this.add(scrollbar);
33
34               this.updateText();
35               this.pack();
36               this.setVisible(true);
37         }
38
39         public void actionPerformed(ActionEvent e){
40                 if (e.getActionCommand().equals("Reset")){
41                         this.scrollbar.setValue(0);
42                         this.updateText();
43                 }
44         }
45
46         public void adjustmentValueChanged(AdjustmentEvent e){
47                 if (e.getSource().equals(scrollbar)){
                        this.updateText();
```

```
48                  }
49          }
50
51          private void updateText(){
52                  status.setText("Scroll Value = " + scrollbar.getValue());
53          }
54
55          public static void main(String[] args) {
56                  new ScrollbarExampleEvents();
57          }
58
59          public void windowActivated(WindowEvent arg0) {
60                  this.scrollbar.setValue(50);
61                  this.updateText();
62          }
63          public void windowClosed(WindowEvent arg0) {}
64          public void windowClosing(WindowEvent arg0) {
65                  System.exit(0);
66          }
67          public void windowDeactivated(WindowEvent arg0) {}
68          public void windowDeiconified(WindowEvent arg0) {}
69          public void windowIconified(WindowEvent arg0) {}
70          public void windowOpened(WindowEvent arg0) {}
71  }
```

Now if you use the 'X' on the top right-hand corner the program exits and if you minimise and then maximise you will see the value reset to 50.

Custom Components

Because we are using Java, which is an object-oriented programming language, we can take any class, extend it with our own functionality or replace some of its functionality or behaviour. We have already met the java.awt.TextField class, which allows you to enter a string in a text field container.

Say we wanted to develop our own TextField (java.awt.TextField) component that only allows you to enter in numeric values (i.e. no letters), but also allow us to press the BACK SPACE key or the DELETE key on the keyboard, how can we do this?

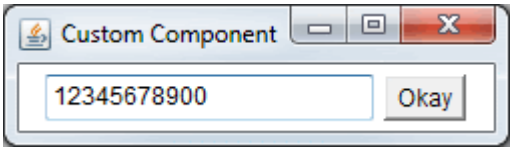


Figure - The Custom Component Application

Well the first step is to extend the java.awt.TextField component and add our custom behaviour in a new class called IntegerTextField that is defined below:

IntegerTextField.java

```
1  package ee402;
2
3  import java.awt.TextField;
4  import java.awt.event.KeyEvent;
5  import java.awt.event.KeyListener;
6
7  @SuppressWarnings("serial")
8  public class IntegerTextField extends TextField implements KeyListener {
9
10         public IntegerTextField(int size){
11                 super(size);
12                 this.addKeyListener(this);
13         }
14
15         public void keyPressed(KeyEvent e) {}
16         public void keyReleased(KeyEvent e) {}
17         public void keyTyped(KeyEvent e) {
18                 char c = e.getKeyChar();
19
20          if(Character.isDigit(c)||c==KeyEvent.VK_DELETE||c==KeyEvent.VK_BACK_SPACE){
21                         System.out.println("Numeric key pressed");
22                 }
23                 else {
24                         System.out.println("Non numeric key pressed");
25                         e.consume();
26                 }
27         }
   }
```

- We extend the TextField class so that the "IntegerTextField is a TextField".
- The KeyListener allows us to capture key press events and requires us to implement the three methods: keyPressed(KeyEvent), keyReleased(KeyEvent) and keyTyped(KeyEvent).
- We can find out which key was pressed within the keyTyped() method by using the e.getKeyChar() method.
- We then need to find out if the char is numeric (0-9) or one of the special keys, which are defined as static constants KeyEvent.VK_DELETE and KeyEvent.VK_BACK_SPACE.
- If it is one of these keys then we will (temporarily) display the message "Numeric key pressed" if not then it must be a non-numeric key.
- If it is a non-numeric key we print the message "Non numeric key pressed" and then we will consume the event using e.consume(), which will stop the event from being passed on for further processing - like cancelling the event.

And then to use this, we build an application as usual:

CustomComponentApp.java

```java
package ee402;

import java.awt.*;
import java.awt.event.*;

@SuppressWarnings("serial")
public class CustomComponentApp extends Frame implements ActionListener,
WindowListener {

	private Button okay;
	private IntegerTextField intField;

	public CustomComponentApp()
	{
	  super("Custom Component");
	  this.setLayout(new FlowLayout());
	  intField = new IntegerTextField(20);

	  this.okay = new Button("Okay");
	  this.okay.addActionListener(this);

	  this.add(intField);
	  this.add(okay);

	  this.pack();
	  this.setVisible(true);
	  this.addWindowListener(this);
	}

	public void actionPerformed(ActionEvent e)
	{
		if (e.getSource().equals(okay)){
			this.intField.setText("0");
		}
	}

	public static void main(String[] args) {
		new CustomComponentApp();
	}

	public void windowActivated(WindowEvent arg0) {}
	public void windowClosed(WindowEvent arg0) {}
	public void windowClosing(WindowEvent arg0) {
		System.exit(0);
	}
	public void windowDeactivated(WindowEvent arg0) {}
	public void windowDeiconified(WindowEvent arg0) {}
	public void windowIconified(WindowEvent arg0) {}
```

```
48          public void windowOpened(WindowEvent arg0) {}
49  }
```

You can see in this example that the new IntegerTextField behaves exactly like a regular TextField, except in the application you cannot press keys other than 0 to 9 and the BACKSPACE and DELETE keys.

Components and Graphics

The Canvas class

As you have previously, there are many components that we can use.

Graphical components have a set of methods that we can override and replace, in order to build components with custom behaviour. Java provides us with the Canvas class that acts like any other component, but also allows you to draw graphics directly to it - just like an artist's canvas. The other advantage of using a component is that you can take advantage of the layout managers to control how the application will place the component as it is moved or resized. The Canvas component specifies its coordinate system at (0,0) for the top left-hand corner.

Canvas components can handle mouse events. You must subclass Canvas to add the behaviour you require, modifying the paint() method. So, for example:

Write a custom Canvas:

```
public class CustomCanvas extends Canvas {

   public void paint(Graphics g){
                  g.drawString("Test", 10, 10);
   }
}
```

Then add it to the Application:

```
public class CanvasApplication extends Frame {

   private CustomCanvas canvas;

   public CanvasApplication(){
...
                  this.canvas = new CustomCanvas();
                  this.add(canvas, BorderLayout.CENTER);
         ...
   }

   public static void main(String[] args) {
                  new CanvasApplication();
   }
}
```

This provides your own Canvas class and you can create as many objects of this as you require - each one only displaying "Test" at the (x,y) location (10,10).

A Functional Canvas Example

Here is an example of an application that uses the Canvas class. It combines a BorderLayout with a Button component and a modified Canvas component. The application generates 200 random circles with different colours every time the "Refresh" button is pressed. Figure 6.X, "The Canvas Application" displays the application running, and you can also run it yourself using the source code below.

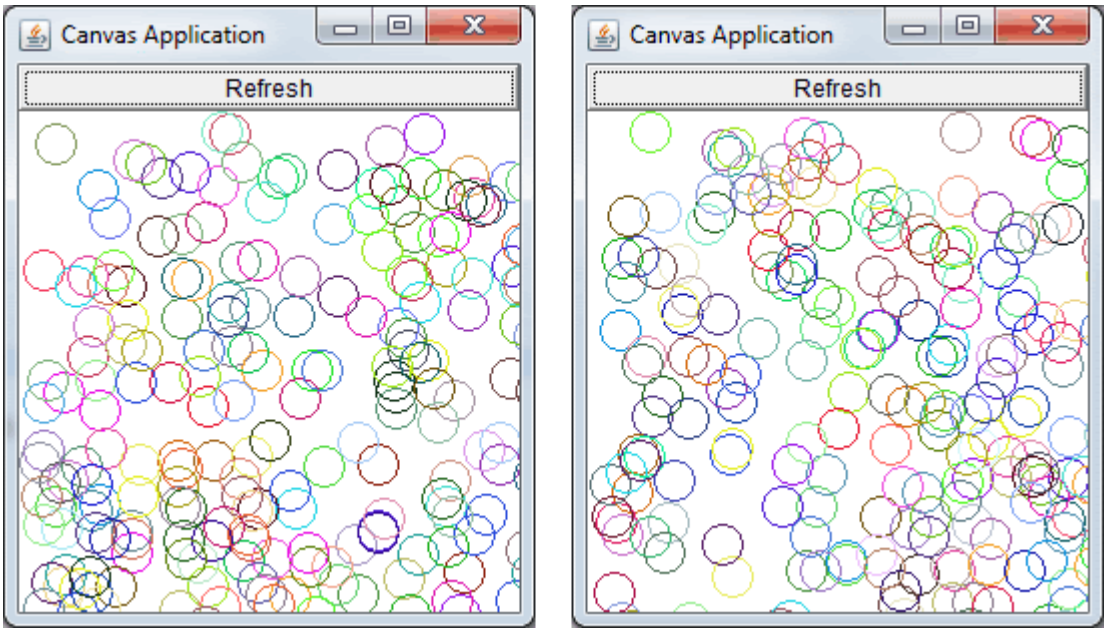


Figure 6.X The Canvas Application with two different random sets of circles

CustomCanvas.java

```java
package ee402;

import java.awt.Canvas;
import java.awt.Color;
import java.awt.Graphics;

@SuppressWarnings("serial")
public class CustomCanvas extends Canvas {

	int width, height;

	public CustomCanvas(int width, int height){
		this.setSize(width,height);
		this.width = width;
		this.height = height;
		this.update();
	}

	public void update() { this.repaint(); }

	public void paint(Graphics g){
		for(int i=0; i<200; i++){
			Color tempColor = new Color((float)Math.random(),
					(float)Math.random(), (float)Math.random());
			g.setColor(tempColor);

g.drawOval((int)(Math.random()*width),(int)(Math.random()*height), 20, 20);
		}
	}
}
```

CanvasApplication.java

```java
package ee402;

import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.WindowEvent;
import java.awt.event.WindowListener;

@SuppressWarnings("serial")
public class CanvasApplication extends Frame implements ActionListener,
WindowListener{

	private CustomCanvas canvas;
	private Button refresh;

	public CanvasApplication(){
		super("Canvas Application");
		this.refresh = new Button("Refresh");
		this.add(refresh, BorderLayout.NORTH);
		this.refresh.addActionListener(this);
		this.canvas = new CustomCanvas(250,250);
		this.add(canvas, BorderLayout.CENTER);
		this.addWindowListener(this);
		this.pack();
		this.setVisible(true);
	}

	public void windowActivated(WindowEvent arg0) {}
	public void windowClosed(WindowEvent arg0) {}
	public void windowClosing(WindowEvent arg0) { System.exit(0); }
	public void windowDeactivated(WindowEvent arg0) {}
	public void windowDeiconified(WindowEvent arg0) {}
	public void windowIconified(WindowEvent arg0) {}
	public void windowOpened(WindowEvent arg0) {}

	public void actionPerformed(ActionEvent e) {
		if(e.getSource().equals(refresh)){
			this.canvas.update();
		}
	}

	public static void main(String[] args) {
		new CanvasApplication();
	}
}
```

In this piece of code I have written two classes CanvasApplication and CustomCanvas. In the CanvasApplication class an object is created of the

CustomCanvas class and this is added to the "Center" of the BorderLayout and added the Button object to the "North". When the button "Refresh" is pressed the update() method of our CustomCanvas is called.

The CustomCanvas class extends Canvas and overwrites the paint() method. The update() just calls this.repaint() which calls the paint() method indirectly, without requiring a Graphics object. To create the Color object we used the constructor:

```
Color(float red, float green, float blue);
```

Where each float has the value 0.0 to 1.0 representing the red, green and blue level. In this case we used Math.random() which generates a double with a value between 0.0 and 1.0. Since it is a double it needs to be converted to a float using the cast conversion. In the drawOval() method the (x,y) position is given by a random number between 0 and 200, by using the Math.random() method and multiplying the value by 200 and then cast converting the double value to an int.

Graphics and Mouse Events

In the previous example I showed how we can add custom graphics to our user interface. However, what if we would like to make these graphics interactive. For example, if we wished to make a graph display, where you could interact with the graph. Well, because a Canvas is-a Component we can add Mouse Events to our custom canvas. Mouse Events are a mechanism for deciding how the mouse should behave when it interacts with our custom canvas - so, what should happen on the custom canvas when you press and release the mouse button.

In this example we would like to draw a red circle around the point that was chosen. In this case the circle will have a radius of 10 pixels.

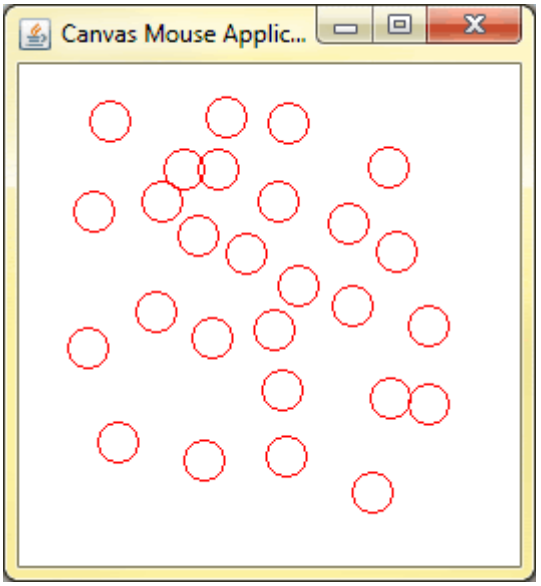


Figure X. Canvas Applications with Events

So, the application remains largely unchanged as below:

CanvasApplicationMouse.java

```java
package ee402;

import java.awt.*;
import java.awt.event.WindowEvent;
import java.awt.event.WindowListener;

@SuppressWarnings("serial")
public class CanvasApplicationMouse extends Frame implements WindowListener{

	private CustomCanvasMouse canvas;

	public CanvasApplicationMouse(){
		super("Canvas Mouse Application");
		this.canvas = new CustomCanvasMouse(250,250);
		this.add(canvas, BorderLayout.CENTER);
		this.addWindowListener(this);
		this.pack();
		this.setVisible(true);
	}

	public void windowActivated(WindowEvent arg0) {}
	public void windowClosed(WindowEvent arg0) {}
	public void windowClosing(WindowEvent arg0) { System.exit(0); }
	public void windowDeactivated(WindowEvent arg0) {}
	public void windowDeiconified(WindowEvent arg0) {}
	public void windowIconified(WindowEvent arg0) {}
	public void windowOpened(WindowEvent arg0) {}

	public static void main(String[] args) {
		new CanvasApplicationMouse();
	}
}
```

But, we add Mouse Events to the Custom Canvas. You can see here that this class now implements MouseListener and has additional methods to describe what happens when: the mouse is clicked; the mouse enters the canvas region; the mouse exits the canvas region; the mouse button is pressed (but not release); and, the mouse button is released (having subsequently been pressed).

CustomCanvasMouse.java

```java
package ee402;

import java.awt.*;
import java.awt.event.*;
import java.util.*;

@SuppressWarnings("serial")
public class CustomCanvasMouse extends Canvas implements MouseListener{

	int width, height;
	Vector<Point> points = new Vector<Point>(10);
	static int radius = 10;

	public CustomCanvasMouse(int width, int height){
		this.setSize(width,height);
		this.width = width;
		this.height = height;
		this.addMouseListener(this);
		this.repaint();
	}

	public void paint(Graphics g){
		g.setColor(Color.red);
		for(int i=0; i<points.size(); i++){
			Point p = points.elementAt(i);
			g.drawOval(p.x, p.y, 2*radius, 2*radius);
		}
	}

	public void mouseClicked(MouseEvent e) {
		points.addElement(new Point(e.getX()-radius, e.getY()-radius));
		this.repaint();
	}
	public void mouseEntered(MouseEvent e) {}
	public void mouseExited(MouseEvent e) {}
	public void mousePressed(MouseEvent e) {}
	public void mouseReleased(MouseEvent e) {}
}
```

This application uses the interface MouseListener, by using the keyword implements. When we use an interface in our code we **are required** to write code for the methods defined in that interface. In this case we are using the interface MouseListener which has five methods that must be implemented mousePressed(), mouseReleased(), mouseEntered(), mouseExited() and mouseClicked(). As you can see in the code segment above, I only really required mouseClicked() as I only wanted a circle to be added when the mouse was clicked. However, I was required

to write code for the four other methods and I did, by writing blank functionality for those methods - so nothing will occur when the other four events occur.

As well as writing the code for the events that occur, we also have to "turn on" Mouse Listening. We do this by using the line of code this.addMouseListener(this); What we are doing is stating "To this CustomCanvasMouse object turn on Mouse Listening - and if a mouse event occurs then send it to this CustomCanvasMouse object". Our custom canvas is now ready to listen for events and pass them to its own five mouse methods.

Finally - one small point. At the top of the code I wrote import java.awt.*; and I also wrote import java.awt.event.*;. This may seem unnecessary but it is not. We need the MouseEvent class from the java.awt.event package for our code, but does the import java.awt.*; not also import the classes in the java.awt.event package? No, it does not! - the * is not recursive and so does not import sub packages (sub directories).

Controlling the Canvas

Application -> Custom Canvas

Now that we can trigger events on the custom canvas, we may also want to build a way of controlling the custom canvas from the broader application. For this example, I am going to use a Choice component that allows you to choose a colour. When you choose the colour, all of the items in the custom canvas will change to that colour (red, green and blue).

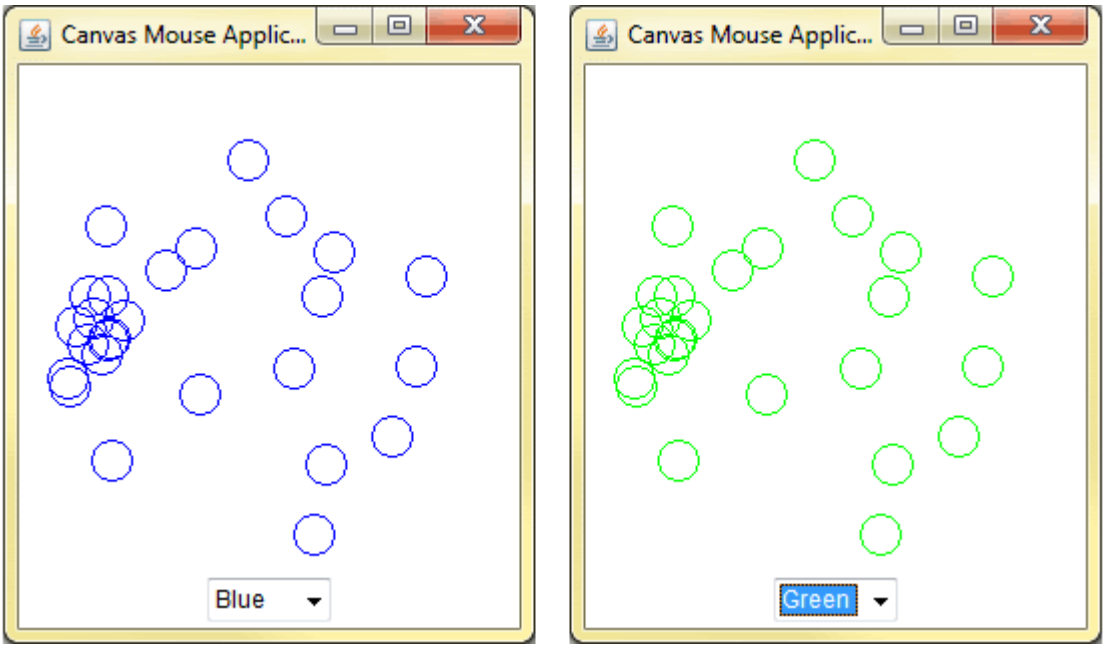


Figure 9.X Examples of the Custom Canvas with Control

The source code of the custom canvas needs to change slightly. I have renamed it to CustomCanvasControl and I have added in a method called setColor() that allows you to pass a Color object from the application to the canvas. Once you pass this Color object, the custom canvas redraws itself with the circles in their new colour. Everything else stays largely the same, except that I now have to share the Color object between the methods, so I have added a Color state called drawColor that allows the Color object to be shared between the setColor() method and the paint() method.

CustomCanvasControl.java

```java
package ee402;

import java.awt.*;
import java.awt.event.*;
import java.util.*;

@SuppressWarnings("serial")
public class CustomCanvasControl extends Canvas implements MouseListener{

	int width, height;
	Color drawColor = Color.red;
	Vector<Point> points = new Vector<Point>(10);
	static int radius = 10;

	public CustomCanvasControl(int width, int height){
		this.setSize(width,height);
		this.width = width;
		this.height = height;
		this.addMouseListener(this);
		this.repaint();
	}

	public void paint(Graphics g){
		g.setColor(drawColor);
		for(int i=0; i<points.size(); i++){
			Point p = points.elementAt(i);
			g.drawOval(p.x, p.y, 2*radius, 2*radius);
		}
	}

	public void setColor(Color c){
		drawColor = c;
		this.repaint();
	 }

	public void mouseClicked(MouseEvent e) {
		points.addElement(new Point(e.getX()-radius, e.getY()-radius));
		this.repaint();
	}
	public void mouseEntered(MouseEvent e) {}
	public void mouseExited(MouseEvent e) {}
	public void mousePressed(MouseEvent e) {}
	public void mouseReleased(MouseEvent e) {}
}
```

The source code of the application needs to change to include the Choice component. I have called this component colorChoice and set it as a state of the class, so that it can be shared between the constructor and the itemStateChanged() method.

This class now needs to implement the ItemListener interface as this is the way that we trap events that occur when you choose one of the colours in the Choice component. This requires us to write a method itemStateChanged() that is called when the chosen item changes. The Choice component is listening for events thanks to the line colorChoice.addItemListener(this) in the constructor.

CanvasApplicationControl.java

```java
package ee402;

import java.awt.*;
import java.awt.event.*;

@SuppressWarnings("serial")
public class CanvasApplicationControl extends Frame implements WindowListener,
ItemListener{

	private CustomCanvasControl canvas;
	private Choice colorChoice;

	public CanvasApplicationControl(){
		super("Canvas Mouse Application");
		this.canvas = new CustomCanvasControl(250,250);
		this.add(canvas, BorderLayout.CENTER);

		Panel south = new Panel(new FlowLayout());
		colorChoice = new Choice();
		colorChoice.addItem("Red");
		colorChoice.addItem("Green");
		colorChoice.addItem("Blue");
		south.add(colorChoice);
		colorChoice.addItemListener(this);
		this.add(south, BorderLayout.SOUTH);

		this.addWindowListener(this);
		this.pack();
		this.setVisible(true);
	}

	public void itemStateChanged(ItemEvent e) {
		if (e.getItem().equals("Red")) canvas.setColor(Color.red);
		else if (e.getItem().equals("Green")) canvas.setColor(Color.green);
		 else canvas.setColor(Color.blue);
	}

	public void windowActivated(WindowEvent arg0) {}
	public void windowClosed(WindowEvent arg0) {}
	public void windowClosing(WindowEvent arg0) { System.exit(0); }
	public void windowDeactivated(WindowEvent arg0) {}
	public void windowDeiconified(WindowEvent arg0) {}
	public void windowIconified(WindowEvent arg0) {}
	public void windowOpened(WindowEvent arg0) {}

	public static void main(String[] args) {
		new CanvasApplicationControl();
	}
```

```
48 }
```

The important point here is that it is quite easy for us to call the setColor() method on our custom canvas, as our custom canvas is a-part-of our main application. We call this method in the itemStateChanged() method and the chosen colour is passed to the custom canvas.

That is the way that we have worked with components so far, and is no different than calling a method on an object after we have created it.

Canvas Application with Callback

Custom Canvas -> Application

So, we have written code to allow our application to modify the state/properties of the custom canvas by calling its methods. The question now is how do we do it the other way? In our application, every time you press the button you create a new circle -- what if we wanted to tell the application how many circles had been created? This is a form of callback, as the application has no idea when a new circle will be created, so we need to pass that information from the custom canvas back to the application.

In the figure below you can see the number of circles displayed in the read-only TextField.

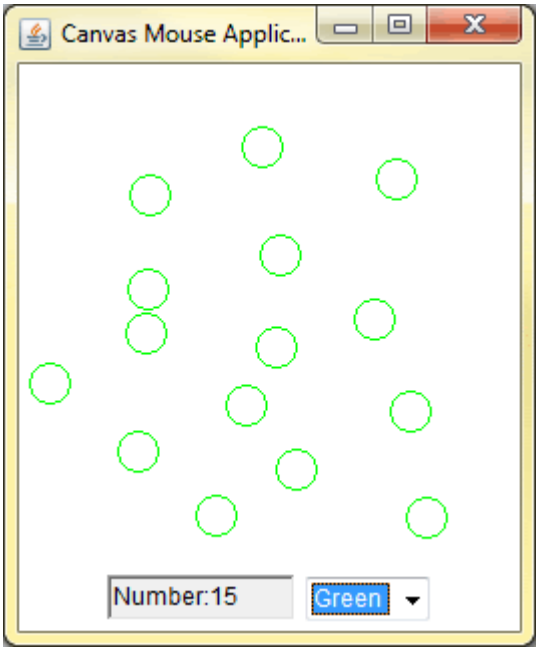


Figure X. Canvas Application with Callback

The application needs to be modified slightly as we now need to pass a reference of the application to the custom canvas. You can see that below - when we create the custom canvas we use this.canvas = new CustomCanvasCallback(this, 250, 250); so we are passing a reference to this object (the application) to the constructor of the CustomCanvas, which has been modified to receive the reference.

We also add a new method in the application called setNumberCircles(int) that allows us to pass the message from the custom canvas to the application. This method simply places the number in the TextField object called numberCircles.

CanvasApplicationCallback.java

```java
package ee402;

import java.awt.*;
import java.awt.event.*;

@SuppressWarnings("serial")
public class CanvasApplicationCallback extends Frame implements WindowListener,
ItemListener{

	private CustomCanvasCallback canvas;
	private Choice colorChoice;
	private TextField numberCircles;

	public CanvasApplicationCallback(){
		super("Canvas Mouse Application");
		this.canvas = new CustomCanvasCallback(this, 250,250);
		this.add(canvas, BorderLayout.CENTER);

		Panel south = new Panel(new FlowLayout());
		numberCircles = new TextField(10);
		numberCircles.setEditable(false);
		this.setNumberCircles(0);
		south.add(numberCircles);
		colorChoice = new Choice();
		colorChoice.addItem("Red");
		colorChoice.addItem("Green");
		colorChoice.addItem("Blue");
		south.add(colorChoice);
		colorChoice.addItemListener(this);
		this.add(south, BorderLayout.SOUTH);


		this.addWindowListener(this);
		this.pack();
		this.setVisible(true);
	}

	public void setNumberCircles(int number){
		this.numberCircles.setText("Number:" + number);
	}

	public void itemStateChanged(ItemEvent e) {
		if (e.getItem().equals("Red")) canvas.setColor(Color.red);
		else if (e.getItem().equals("Green")) canvas.setColor(Color.green);
		else canvas.setColor(Color.blue);
	}

	public void windowActivated(WindowEvent arg0) {}
```

```
48          public void windowClosed(WindowEvent arg0) {}
49          public void windowClosing(WindowEvent arg0) { System.exit(0); }
50          public void windowDeactivated(WindowEvent arg0) {}
51          public void windowDeiconified(WindowEvent arg0) {}
52          public void windowIconified(WindowEvent arg0) {}
53          public void windowOpened(WindowEvent arg0) {}
54
55          public static void main(String[] args) {
56                  new CanvasApplicationCallback();
57          }
58  }
```

The custom canvas object has been modified below to now receive a reference to the calling application in the constructor, so you can see "CanvasApplicationCallback app" being passed to the canvas object. Now that we have this reference (called callingApp), we are now able to call the setNumberCircles(int) method. So, when we redraw the canvas we can send a message back to the application to tell it to update the text in the TextField (even if it doesn't need to be updated).

CustomCanvasCallback.java

```java
package ee402;

import java.awt.*;
import java.awt.event.*;
import java.util.*;

@SuppressWarnings("serial")
public class CustomCanvasCallback extends Canvas implements MouseListener{

	int width, height;
	Color drawColor = Color.red;
	Vector<Point> points = new Vector<Point>(10);
	static int radius = 10;
	CanvasApplicationCallback callingApp;

    public CustomCanvasCallback(CanvasApplicationCallback app, int width,
            int height){
		this.setSize(width,height);
		this.width = width;
		this.height = height;
		this.addMouseListener(this);
		this.callingApp = app;
		this.repaint();
	}

	public void paint(Graphics g){
		g.setColor(drawColor);
		for(int i=0; i<points.size(); i++){
			Point p = points.elementAt(i);
			g.drawOval(p.x, p.y, 2*radius, 2*radius);
		}
		this.callingApp.setNumberCircles(points.size());
	}

	public void setColor(Color c){
		drawColor = c;
		this.repaint();
	}

	public void mouseClicked(MouseEvent e) {
		points.addElement(new Point(e.getX()-radius, e.getY()-radius));
		this.repaint();
	}
	public void mouseEntered(MouseEvent e) {}
	public void mouseExited(MouseEvent e) {}
	public void mousePressed(MouseEvent e) {}
	public void mouseReleased(MouseEvent e) {}
}
```

Properly with Java Interfaces

Custom Canvas -> Application using Interfaces

The previous example works very well; however, there is one major flaw. If we want to re-use our custom canvas in another application called DrawingApp then we can't without changing the code, as it requires an application called CanvasApplication to be passed. Clearly we need a way to allow the name of the calling class to change or our code will not be very portable.

To do this, we use Java Interfaces. We saw Java Interfaces before, but what they allow us to do here is to abstract the calling application to "any application that implements our Java interface"

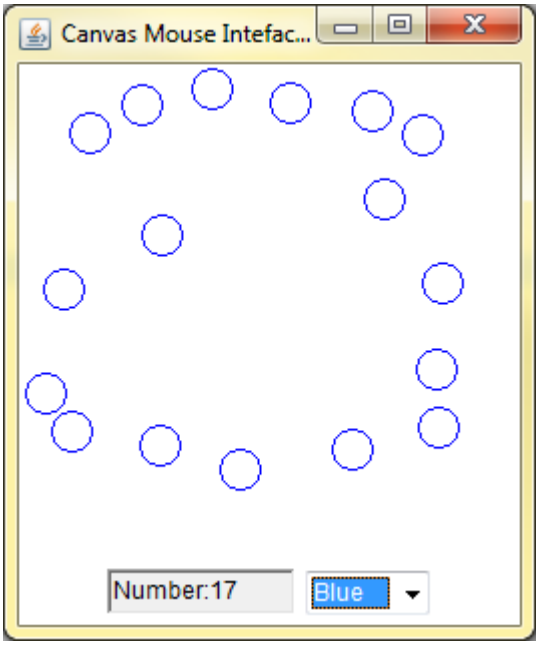


Figure X. Canvas Mouse Application with Interfaces

So, here is our Java interface. At the moment there is only one method that we allow our custom canvas to call on the main application and we'll rename it slightly to setNumber(int). So, this interface below states that if any class implements the CounterField interface then it must write the code for a method called setNumber. So, our class can have any name!

CounterField.java

```java
package ee402;

public interface CounterField {
        public void setNumber(int number);
}
```

So, here is the application implementing the CounterField interface. Without reading past the "public class CanvasApplication..." line, I know that there must be a setNumber(int) method, otherwise the code could not compile.

CanvasApplicationInterfaces.java

```java
package ee402;

import java.awt.*;
import java.awt.event.*;

@SuppressWarnings("serial")
public class CanvasApplicationInterfaces extends Frame implements WindowListener,
ItemListener, CounterField{

	private CustomCanvasInterfaces canvas;
	private Choice colorChoice;
	private TextField numberCircles;

	public CanvasApplicationInterfaces(){
		super("Canvas Mouse Intefaces");
		this.canvas = new CustomCanvasInterfaces(this, 250,250);
		this.add(canvas, BorderLayout.CENTER);

		Panel south = new Panel(new FlowLayout());
		numberCircles = new TextField(10);
		numberCircles.setEditable(false);
		this.setNumber(0);
		south.add(numberCircles);
		colorChoice = new Choice();
		colorChoice.addItem("Red");
		colorChoice.addItem("Green");
		colorChoice.addItem("Blue");
		south.add(colorChoice);
		colorChoice.addItemListener(this);
		this.add(south, BorderLayout.SOUTH);

		this.addWindowListener(this);
		this.pack();
		this.setVisible(true);
	}

	public void setNumber(int number){
		 this.numberCircles.setText("Number:" + number);
	}

	public void itemStateChanged(ItemEvent e) {
		if (e.getItem().equals("Red")) canvas.setColor(Color.red);
		else if (e.getItem().equals("Green")) canvas.setColor(Color.green);
		else canvas.setColor(Color.blue);
	}

	public void windowActivated(WindowEvent arg0) {}
	public void windowClosed(WindowEvent arg0) {}
```

```
48          public void windowClosing(WindowEvent arg0) { System.exit(0); }
49          public void windowDeactivated(WindowEvent arg0) {}
50          public void windowDeiconified(WindowEvent arg0) {}
51          public void windowIconified(WindowEvent arg0) {}
52          public void windowOpened(WindowEvent arg0) {}
53
54          public static void main(String[] args) {
55                  new CanvasApplicationInterfaces();
56          }
57  }
```

We pass the object of CanvasApplicationInterfaces directly to the constructor of the custom canvas class and it looks like the same as before; however, there is one major difference, which can be seen below - The custom canvas does not expect an object of CanvasApplicationInterfaces, it expects an object of the interface CounterField (see the constructor).

CustomCanvasInterfaces.java

```java
package ee402;

import java.awt.*;
import java.awt.event.*;
import java.util.*;

@SuppressWarnings("serial")
public class CustomCanvasInterfaces extends Canvas implements MouseListener{

	int width, height;
	Color drawColor = Color.red;
	Vector<Point> points = new Vector<Point>(10);
	static int radius = 10;
	CounterField callingApp;

	public CustomCanvasInterfaces(CounterField app, int width, int height){
		this.setSize(width,height);
		this.width = width;
		this.height = height;
		this.addMouseListener(this);
		this.callingApp = app;
		this.repaint();
	}

	public void paint(Graphics g){
		g.setColor(drawColor);
		for(int i=0; i<points.size(); i++){
			Point p = points.elementAt(i);
			g.drawOval(p.x, p.y, 2*radius, 2*radius);
		}
		this.callingApp.setNumber(points.size());
	}

	public void setColor(Color c){
		drawColor = c;
		this.repaint();
	}

	public void mouseClicked(MouseEvent e) {
		points.addElement(new Point(e.getX()-radius, e.getY()-radius));
		this.repaint();
	}
	public void mouseEntered(MouseEvent e) {}
	public void mouseExited(MouseEvent e) {}
	public void mousePressed(MouseEvent e) {}
	public void mouseReleased(MouseEvent e) {}
}
```

This means that any class that has implemented the CounterField interface can call the constructor above as it must have the required setNumber() method that is called in the paint() method above.

This is good programming practice as this one Custom Canvas class can be used by many different calling applications, even within the same project.