Introduction

Modern computer operating systems include presentation services that provide the user with a GUI (Graphic User Interface). As a result, computer users now demand application programs that also provide GUIs. Fortunately, the Java platform provides extensive support for building them. GUIs display in graphical form on the screen. To program a GUI, you must master event-driven programming because GUI-oriented programs are event-driven.

In this module, you will learn how to use the most common user interface components in the Java Swing package. In the later section of this module, we also introduce how to write Java programs that interact with disk files and other sources of bytes and characters.

Objectives

- To use inheritance to customize frames
- To understand how user-interface components are added to a container
- To understand the use of layout managers to arrange user interface components in a container
- To become familiar with common user interface components, such as buttons, combo boxes, text areas and menus
- To build programs that handle events from user interface components
- To be able to read and write text files
- To become familiar with the concepts of text and binary file formats

Terminology of a GUI

All interactive output and input should pass through your program’s GUI. Therefore, the GUI consists of what the user sees on the screen and code to process user actions such as clicking the mouse or typing on the keyboard. Your program must properly relate those actions to the elements displayed in the GUI and perform the appropriate activities in response.

Components comprise a major part of a GUI. In the Java platform, components are predefined standard elements such as buttons, text fields, frame windows, and dialog boxes. The Swing and AWT APIs provide a large repertoire of components.

The display space on the screen is also a component. A Java application window is a frame window. Frame windows have a title and a border; buttons for closing, minimizing, and maximizing the window; and can contain a menu bar. All your controls (such as check boxes, labels, text areas, etc.) need to be put into the frame to display them. Components like the frame and panels are also called containers.
Using Inheritance to Customize Frames

For building a complex GUI with many interface components added to a frame, you usually start by inheriting the JFrame class. The JFrame (belongs to the javax.swing) class defined and inherits many useful fields and methods. The most commonly used methods are, setSize and setVisible. The class InvestmentFrame and class InvestmentFrameViewer on page 516 and page 517 of the text demonstrate how to inherit the JFrame class in your application. The approach is to let your GUI extend JFrame class, and then to create an object in your GUI with the type JFrame. Another common seen style can be found in Advanced Topics 14.1 (on page 517 of text).

Layout management

Components are arranged by placing them inside containers. Each container has a layout manager that directs the arrangement of its components. Three useful layout managers are the border layout, the flow layout and the grid layout.

Border layout – Groups the container into five areas: centre, north, west, south and east. When adding a component to a container with the border layout, specify the NORTH, EAST, SOUTH, WEST or CENTRE position to specify where the component lies in relation to other components.

Read and evaluate the following example.

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

public class TestLayout
{
    public static void main(String[] args)
    {
        JFrame f = new JFrame("Test Layout");
        JPanel p = new JPanel();
        p.setLayout(new BorderLayout());

        JButton up = new JButton("UP");
        JButton down = new JButton("DOWN");
        JButton left = new JButton("LEFT");
        JButton right = new JButton("RIGHT");

        p.add(up,BorderLayout.NORTH);
        p.add(down,BorderLayout.SOUTH);
        p.add(left,BorderLayout.WEST);
        p.add(right,BorderLayout.EAST);

        f.add(p);
        f.setSize(FRAME_WIDTH, FRAME_HEIGHT);
        f.setVisible(true);
    }

    private static final int FRAME_WIDTH=200;
    private static final int FRAME_HEIGHT=200;
}
```
Flow layout – A flow layout simply arranges its components from left to right and starts a new row when there is no more room in the current row. A JPanel uses a flow layout by default.

```
p.setLayout(new FlowLayout());
JButton A = new JButton("ACT");
JButton B = new JButton("QLD");
JButton C = new JButton("NSW");
JButton D = new JButton("VIC");
JButton E = new JButton("TAS");
JButton F = new JButton("NT");
JButton G = new JButton("WA");
p.add(A);
p.add(B);
p.add(C);
p.add(D);
p.add(E);
p.add(F);
p.add(G);
```

Grid layout – The grid layout arranges components in a grid with a fixed number of rows and columns, resizing each of the components so that they all have the same size.

```
p.setLayout(new GridLayout(4,3));
JButton n1 = new JButton("1");
JButton n2 = new JButton("2");
JButton n3 = new JButton("3");
JButton n4 = new JButton("4");
JButton n5 = new JButton("5");
JButton n6 = new JButton("6");
JButton n7 = new JButton("7");
JButton n8 = new JButton("8");
JButton n9 = new JButton("9");
JButton n0 = new JButton("0");
JButton ndot = new JButton("." winding if nothing is entered;
JButton nce = new JButton("CE");
p.add(n7);
p.add(n8);
p.add(n9);
p.add(n4);
p.add(n5);
p.add(n6);
p.add(n1);
p.add(n2);
p.add(n3);
p.add(n0);
p.add(ndot);
p.add(nce);
```
The above three layout managers are most commonly used layouts. You can create acceptable-looking layouts in nearly all situations by nesting panels. You give each panel an appropriate layout manager. Panels don't have visible borders, so you can use as many panels as you need to organize your components.

**Choices (Radio Buttons, Check Boxes and Combo boxes)**

**Radio buttons** work like the station selector buttons on a car radio: If you select a new station, the old station is automatically deselected. You add radio buttons into a ButtonGroup so that only one button in the group is on at any time. The following screen shot is an example of RadioButtons and its code.

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

public class TestLayout
{
    public static void main(String[] args)
    {
        JFrame f = new JFrame("TestLayout");
        JPanel p = new JPanel();
        p.setLayout(new GridLayout(1,3));

        JRadioButton n1 = new JRadioButton("Small");
        JRadioButton n2 = new JRadioButton("Medium");
        JRadioButton n3 = new JRadioButton("Large");

        ButtonGroup g = new ButtonGroup();
        g.add(n1);
        g.add(n2);
        g.add(n3);

        p.add(n1);
        p.add(n2);
        p.add(n3);

        f.add(p);
        f.setSize(FRAME_WIDTH, FRAME_HEIGHT);
        f.setVisible(true);
    }

    private static final int FRAME_WIDTH=400;
    private static final int FRAME_HEIGHT=100;
}
```
Note, you **cannot** add a button group to a panel directly. A panel has to have each individual radio button added on to it.

A **Check box** is a user interface component with two states: checked and unchecked. You use a group of check boxes when one selection does not exclude another. That mean, a user can select more than one value. The following screen shot is a check box and its corresponding code pieces.

```
setLayout(new GridLayout(1,3));
JCheckBox n1 = new JCheckBox("Rocky");
JCheckBox n2 = new JCheckBox("Mackay");
JCheckBox n3 = new JCheckBox("Bundy");
JCheckBox n4 = new JCheckBox("Glandston");
p.add(n1);
p.add(n2);
p.add(n3);
p.add(n4);
```

Note: you do not need to group check boxes. Each individual check box can be directly added to the panel.

A **Combo Box** is a drop down selection list. To create a combo box and add items to the list, use:

```
JComboBox state = new JComboBox( );
state.addItem("NSW");
state.addItem("QLD");
state.addItem("VIC");
```

Note : you can add ComboBox into a panel, e.g.

```
JPanel p = new JPanel( );
p.add(state);
```

Radio buttons, check boxes, and combo boxes generate action events, just as buttons do. You need to add action listeners and implement actionPerformed method when necessary. The programs ChoiceFrameViewer.java and ChoiceFrame.java (page 523, 524 of the text) demonstrate how to use these three components in a GUI.
Activity
Heading “How To 14.1” on pages 528, 529, 530, Layout Management, is very important for a GUI programming. It is helpful for you to start your assignment 2 as well. You are advised to read through this part carefully, and apply the 5 steps listed in this section in your assignment.

Menus
A frame contains a menu bar. The menu bar contains menus. A menu contains submenus and menu items. Only menu items generate action events. The sequence normally used is to:

- create a menu bar container (to hold menus)
- create a menu container for each menu (to hold menu items)
- create menu items and handle events for each menu item

When your menu has more levels, the code can become very long.
You need to read and understand the programs on pages 531-535. This is a good example for how to construct a menus.

Text Area
Use a JTextArea to show multiple lines of text. Note this is different from a JTextField.
For example,

```java
final int ROWS = 10;
final int COLUMNS = 30;
JTextArea t = new JTextArea(ROWS, COLUMNS);
```

The common used methods for a JTextArea are: setText, append, setEditable etc.
You may need to inspect the methods in the API reference when needed.

I/O Stream Overview
(The follow paragraphs are from http://java.sun.com/docs/books/tutorial/essential/io/overview.html)
“To bring in information, a program opens a stream on an information source (a file, memory, a socket) and reads the information sequentially, as shown here:

Similarly, a program can send information to an external destination by opening a stream to a destination and writing the information out sequentially, like this:

No matter where the data is coming from or going to and no matter what its type, the algorithms for sequentially reading and writing data are basically the same:

<table>
<thead>
<tr>
<th>Reading</th>
<th>Writing</th>
</tr>
</thead>
<tbody>
<tr>
<td>open a stream</td>
<td>open a stream</td>
</tr>
<tr>
<td>while more information</td>
<td>while more information</td>
</tr>
<tr>
<td>read information</td>
<td>write information</td>
</tr>
<tr>
<td>close the stream</td>
<td>close the stream</td>
</tr>
</tbody>
</table>

The java.io package contains a collection of stream classes that support these algorithms for reading and writing. To use these classes, a program needs to import the java.io package. The stream classes are divided into two class hierarchies, based on the data type (either characters or bytes) on which they operate.”

Character Streams are used for reading / writing text format information, such as reading / writing information to files. Byte Streams are typically used to read and write binary data such as images and sounds. Both Character Streams and Byte Streams have their own class hierarchies. These class hierarchies can be found from Java I/O API and the above tutorial link. We will introduce a few common used classes for text input and output in the next topic.

You are not required to read / write binary information by using Byte Streams in this course. However, you should know where to find these classes when you dealing with image or sounds I/O.

**Reading and Writing Text Files**

The most common task is to read and write files that contain text. The files are normally created by a text editor such as MS Notepad or TextPad. The class FileReader and FileWriter are the two classes in Character Streams that you need to use to read and write text files. Before Java 5, Java’s input/output mechanisms were awkward. Java 5’s Scanner class makes the input/output to the files and to the console a lot easier.
Reading files
Firstly, let’s see how to use this Scanner class to read a text file.

- Firstly, you need to create an object instance of FileReader (this is to open the stream), it takes a text file name as a parameter, for example:

  ```java
  FileReader reader = new FileReader (“input.txt”);
  ```

- Then, you need to create an object instance of the Scanner class, the Scanner class take a FileReader object as a parameter, that is:

  ```java
  Scanner in = new Scanner (reader);
  ```

- Then you are able to use methods defined in the Scanner class to read text from the file input.txt, for example:

  ```java
  public String next ( ) – read a String has no space
  public String nextLine ( ) -- read a whole line of text, return a String
  public int nextInt ( ) – read next integer
  public double nextDouble ( ) – read next double
  ```

Writing files
When writing text files, use the FileWriter class or PrintWriter class and their methods.

- Firstly, you need to create an object instance of FileWriter or PrintWriter (this is like opening the stream), it takes a text file name as a parameter, for example:

  ```java
  FileWriter out = new FileWriter ( “output.txt” );  or
  PrintWriter out = new PrintWriter ( “output.txt” );
  ```

- Then you are able to use methods defined in these two classes to write text to the file output.txt.

  If using a FileWriter object, use the write method, e.g.

  ```java
  out.write(100);
  out.write(“This is a text file”);
  ```

  If using the PrintWriter object use print or println methods, e.g.

  ```java
  out.print(100);
  out.println(“This is a text file”);
  ```
An example using reading/writing text data from/to files can be found on page 577 of the text. You may need to inspect this code.

**Text and Binary Formats**

In **text** data format, data items are represented in human-readable form such as a sequence of characters: “Hello, World!”. **Character Streams** look after text data format input/output. The **Reader** and **Writer** classes and their subclasses can be used for text data input/output. Use FileReader to read data from files. Use FileWriter or PrintWriter to write data to files.

In **byte** data format, data items are represented by bytes which are not readily understandable by us. **Bytes Streams** look after byte data format input/output.

**Reading**

Text book:
  - Chapter 14: Graphic User Interface (whole chapter required)
  - Chapter 16: Files and Streams, only 16.1 and 16.2 required

**Review questions**

Review exercises:
  - Page 602: Exercise R16.2, R16.4

Programming exercises:
  - Page 548: Exercise R14.3
  - Page 603: Exercise P16.1

**References**

import java.awt.BorderLayout;
import java.awt.GridLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JTextField;

/**
   This frame contains a panel that displays buttons
   for a calculator and a panel with a text fields to
   specify the result of calculation.
*/
public class CalculatorFrame extends JFrame
{
   public CalculatorFrame()
   {
      JPanel calPanel = new JPanel();
      calPanel.setLayout(new GridLayout(4, 4));
      createControlPanel();
      pack();
   }
   
   /**
      Creates the control panel with the text field
      and buttons on the frame.
   */
   private void createControlPanel()
   {
      // the panel for holding the user interface components
      JPanel buttonPanel = new JPanel();
      buttonPanel.setLayout(new GridLayout(4, 4));

      // add buttons
      buttonPanel.add(makeDigitButton("7"));
      buttonPanel.add(makeDigitButton("8"));
      buttonPanel.add(makeDigitButton("9"));
      buttonPanel.add(makeOperatorButton("/"));
      buttonPanel.add(makeDigitButton("4"));
      buttonPanel.add(makeDigitButton("5"));
      buttonPanel.add(makeDigitButton("6"));
      buttonPanel.add(makeOperatorButton("*"));
      buttonPanel.add(makeDigitButton("1"));
      buttonPanel.add(makeDigitButton("2"));
      buttonPanel.add(makeDigitButton("3"));
      buttonPanel.add(makeOperatorButton("-"));
      buttonPanel.add(makeDigitButton("0"));
      buttonPanel.add(makeDigitButton("."));
      buttonPanel.add(makeOperatorButton("="));
      buttonPanel.add(makeOperatorButton("+"));

      display = new JTextField("0");
      display.setEditable(false);
   }
}
add(display, BorderLayout.NORTH);
add(buttonPanel, BorderLayout.CENTER);

lastValue = 0;
lastOperator = "+";
start = true;
}

/**
 Calculates the resulting value in a calculator.
 @param n the number to calculate
 */
public void calculate(double n)
{
    if (lastOperator.equals("+"))
        lastValue += n;
    else if (lastOperator.equals("-"))
        lastValue -= n;
    else if (lastOperator.equals("*"))
        lastValue *= n;
    else if (lastOperator.equals("/"))
        lastValue /= n;
    else if (lastOperator.equals("="))
        lastValue = n;

display.setText("" + lastValue);
}

/**
 Makes a button representing a digit of a calculator.
 @param digit the digit of the calculator
 @return the button of the calculator
 */
public JButton makeDigitButton(final String digit)
{
    JButton button = new JButton(digit);
    class ButtonListener implements ActionListener
    {
        public void actionPerformed(ActionEvent event)
        {
            if (start)
                display.setText(digit);
            else
                display.setText(display.getText() + digit);
            start = false;
        }
    }
    ButtonListener listener = new ButtonListener();
    button.addActionListener(listener);
    return button;
}

/**
 Makes a button representing an operator of a calculator.
 @param op the operator of the calculator
 @return button the button of the calculator
 */
public JButton makeOperatorButton(final String op)
{
    JButton button = new JButton(op);
}
class ButtonListener implements ActionListener
{
    public void actionPerformed(ActionEvent event)
    {
        if (!start)
        {
            double x = Double.parseDouble(display.getText());
            calculate(x);
            start = true;
        }
        lastOperator = op;
    }
}

ButtonListener listener = new ButtonListener();
button.addActionListener(listener);
return button;
}

private JTextField display;
private double lastValue;
private String lastOperator;
private boolean start;

import javax.swing.JFrame;

public class ExP14_3Viewer
{
    public static void main(String[] args)
    {
        JFrame frame = new CalculatorFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setTitle("ExP14_3");
        frame.setVisible(true);
    }
}
import java.io.FileReader;
import java.io.IOException;
import java.util.Scanner;

/**
   * This class tests the FileCounter class.
   */
public class ExP16_1
{
    public static void main(String[] args)
    {
        Scanner in = new Scanner(System.in);
        String input = "";

        try
        {
            boolean done = false;
            while (!done)
            {
                System.out.print("Filename (-1 to end): ");
                input = in.next();
                if (input.equals("-1"))
                    done = true;
                else
                {
                    FileReader fin = new FileReader(input);
                    FileCounter fileCounter = new FileCounter();
                    fileCounter.read(fin);
                    System.out.println(
                        fileCounter.getCharacterCount() + " chars, " +
                        fileCounter.getWordCount() + " words, " +
                        fileCounter.getLineCount() + " lines");
                    fin.close();
                }
            }
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
    }
}
```java
import java.io.FileReader;
import java.io.IOException;

/**
 * A class to count the number characters, words, and lines in a file.
 */
public class FileCounter
{
    /**
     * Construct a FileCounter object.
     */
    public FileCounter()
    {
        chars = 0;
        words = 0;
        lines = 0;
    }

    /**
     * Read the input file and count the number.
     * of words, characters, and lines.
     * @param file the file to count
     */
    public void read(FileReader reader) throws IOException
    {
        boolean space = true;
        boolean more = true;
        while (more)
        {
            int ic = reader.read();
            if (ic == -1) more = false;
            else
            {
                chars++;
                char c = (char) ic;
                if (c == '\n')
                {
                    lines++;
                    space = true;
                }
                else if (c == ' ' || c == '\r')
                    space = true;
                else
                {
                    if (space)
                        words++;
                    space = false;
                }
            }
            lines++;
        }
    }

    /**
     * Gets the number of characters in a file.
     * @return the number of characters
     */
    public long getChars() {
        return chars;
    }
    ...
```
public int getCharacterCount()
{
    return chars;
}

/**
 * Gets the number of words in a file.
 * @return the number of words
 */
public int getWordCount()
{
    return words;
}

/**
 * Gets the number of lines in a file.
 * @return the number of lines
 */
public int getLineCount()
{
    return lines;
}

private int chars;
private int words;
private int lines;