

**MT201**

## ***Unit 9***

---

### **Graphical user interface (GUI)**

**Course team**

Developer: Herbert Shiu, Consultant

Designer: Dr Rex G Sharman, OUHK

Coordinator: Kelvin Lee, OUHK

Member: Dr Vanessa Ng, OUHK

**External Course Assessor**

Professor Jimmy Lee, Chinese University of Hong Kong

**Production**

ETPU Publishing Team

Copyright © The Open University of Hong Kong, 2003.  
Reprinted 2005.

All rights reserved.

No part of this material may be reproduced in any form  
by any means without permission in writing from the  
President, The Open University of Hong Kong.  
The Open University of Hong Kong  
30 Good Shepherd Street  
Ho Man Tin, Kowloon  
Hong Kong

# Contents

---

<b>Introduction</b>	<b>1</b>
<b>Objectives</b>	<b>3</b>
<b>Creating a cross-platform graphical user interface</b>	<b>4</b>
Abstract Window Toolkit (AWT)	5
Swing	6
<b>Creating a simple graphical user interface</b>	<b>8</b>
The steps in creating a graphical user interface	8
A few GUI components: frame, button and label	9
<b>Arranging GUI components with layout managers</b>	<b>23</b>
What are layout managers?	24
The FlowLayout layout manager	26
The BorderLayout layout manager	27
The GridLayout layout manager	29
<b>Other GUI components</b>	<b>33</b>
Text fields	33
Text areas	34
Accessing text fields and text areas	37
Canvas	38
Checkboxes	40
Radio boxes	42
Menu bars, menus and menu items	45
Dialogs	48
<b>Creating complex GUIs</b>	<b>49</b>
Another container: JPanel	49
Complex GUIs with panels	49
<b>Event-handling model</b>	<b>53</b>
Event sources, events and event listeners	53
Interface	55
Implementing event handlings	56
Action events	58
<b>Java applets</b>	<b>68</b>
What are Java applets?	68
Developing a simple Java applet	69
<b>Summary</b>	<b>76</b>

<b>Appendix A: Using labels for showing operation results</b>	<b>79</b>
<b>Appendix B: Further discussion of the three layout manager classes</b>	<b>82</b>
Resizing behaviours of different layout managers	82
Other properties of layout managers	87
<b>Appendix C: On the use of JOptionPane</b>	<b>89</b>
<b>Appendix D: Dialogs</b>	<b>92</b>
<b>Appendix E: Further discussion of Java interfaces</b>	<b>98</b>
<b>Appendix F: Window and mouse category events</b>	<b>105</b>
Window events	105
Mouse events	113
<b>Appendix G: A simple calculator</b>	<b>116</b>
<b>Appendix H: A drawing program</b>	<b>123</b>
<b>Appendix I: Showing Java consoles of Web browsers</b>	<b>130</b>
Netscape	130
Microsoft Internet Explorer	131
<b>Appendix J: Writing a GUI-based payroll software application</b>	<b>133</b>
Introduction to the GUI-based payroll software application	133
The implementation of the software	135
Conclusion	158
<b>References</b>	<b>159</b>
<b>Suggested answers to self-test questions</b>	<b>160</b>

# Introduction

From *Unit 3* to *Unit 7*, you learned software application development using an object-oriented approach with the Java programming language. *Unit 7* is the most important, and hence the longest, unit of all. It introduced you to three key concepts of object-oriented programming — encapsulation, inheritance and polymorphism. In *Unit 8* to *Unit 10*, you learn particular aspects of software development. For example, you learned in *Unit 8* how to use the classes provided by the Java software library for performing input/output operations in a software application.

In this unit, you will learn another aspect of software development — building graphical user interfaces (GUIs). Due to the popularity of windows systems such as Microsoft Windows, computer users have become familiar with GUI-based software applications such as Microsoft Word and Internet Explorer. GUI-based software applications are usually relatively easy to use, and the users take less time to learn how to use them.

With respect to Java programming with GUIs, you have come across the class `JOptionPane` since *Unit 4* for getting user inputs and showing program results with a graphical dialog. However, the functionalities of software applications that use dialogs only are too restrictive. The users expect all functionalities provided by a GUI-based software application to be available to them through menus or buttons so that they can operate the different parts of the application in their own desired ways and order. In this unit, you will learn how to create such types of GUI-based software application.

The Java software library provides a rich set of classes for building GUIs. Then, by making use of these classes in an object-oriented way, you can easily create your own GUIs. We will analyse a GUI using an object-oriented approach, such as deriving ‘is a’ and ‘has a’ relationships among the objects involved. You will see that what you have learned in *Unit 7* is applicable to GUI-based application software.

A GUI-based software application usually shows a standalone window that provides buttons, text-fields, checkboxes and the like on the screen for getting user input and showing operational results. These widgets are known as *GUI components*, and you will learn some common GUI components and their uses in this unit.

After you have learned how to build a GUI-based software application, the application will still be unresponsive in that it will not respond to any stimulation from the users, such as clicking a button or clicking the <Close> button of the window. These ‘stimulations’ are known as *events* in the Java programming language, and you will learn how to equip the GUI-based software application with the ability to handle such events. During the discussion of event handling, you will learn a variant of abstract classes that you learned in *Unit 7* — the Java interfaces.

At the end of this unit, we discuss how to create Java applets, which are the small programs to be executed by a Web browser such as Microsoft Internet Explorer. Then, you can write Java applets to be embedded in Web pages and downloaded to Web browsers for execution.

This unit is interesting, challenging and rewarding, because you will create different GUIs, from simple to complicated ones. Furthermore, we will add GUIs to the payroll calculation software applications we built in previous units, so that it becomes GUI-based and you will appreciate that the source codes (class definitions) written in an object-oriented programming language can easily be reused and extended. The complete discussion on the GUI-based payroll calculation can be found in Appendix J. It is recommended that you read it after you have completed the entire core part of the unit.

# Objectives

By the end of *Unit 9*, you should be able to:

- 1 *Explain* the problem of creating a cross-platform graphical user interface.
- 2 *Describe* the Java way of creating GUIs.
- 3 *Use* different GUI components.
- 4 *Apply* the different layout managers.
- 5 *Develop* complex GUIs with panels.
- 6 *Describe* the concept of event handling.
- 7 *Apply* event handling in GUIs.
- 8 *Apply* Java interfaces.
- 9 *Develop* Java applets.

## Creating a cross-platform graphical user interface

An interface usually refers to the boundary between two entities, and there are interactions between the two. Human beings interact with computers through human-computer interfaces, which usually consist of some hardware for input to the computer and some display for output from it. A software application sits between the user and the computer so that it interactively gets the user inputs for the software to process and shows the operational results afterwards.

Due to the limited computation and display capabilities, many of the computer user interfaces before the mid-1980s were character-based (or text-based). The users entered commands or pressed function keys on the keyboards to operate a software application. As a result, the users had to memorize the commands or the keystrokes for a particular software application, and most users experienced a steep learning curve.

The improvements in computation power and graphical display capability enabled software developers to develop a graphical environment for a particular operating system so that the users could operate the machines more easily. Furthermore, the introduction of the computer mouse provided an alternative and handy way to operate the machines.

A software application or system that provides the above-mentioned graphical environment is usually known as a windowing system. For example, X-Windows and Microsoft Windows are the major windowing systems for UNIX machines and personal computers respectively. By using the software libraries that come with the windowing system, software developers can write their own GUI-based software applications so that the users can operate the software applications as if they are part of the windowing system.

Before the introduction of the Java programming language and JVM, GUI-based applications for X-Windows and Microsoft Windows were mostly written in the C (or C++) programming language and Microsoft Visual Basic (or Visual C++) respectively. As the software libraries for developing GUI-based applications for different windowing systems differ greatly, it is almost impossible to write a GUI-based application to be executed in different windowing systems, without modifications. As a result, it is necessary to customize the software application so that it can be executed in another windowing system. It could be a nightmare to maintain the programs for different operating systems and windowing systems.

As a software application written in the Java programming language is a cross-platform programming language, its software library is designed so that GUI-based application software written in the Java programming language can work with different windowing systems as well. Then, it is certain that if a software application can be executed in a particular windowing system, it can be executed in all other windowing systems in which a corresponding JVM is available. You will see that developing



GUI-based applications with the Java programming language is not concerned with the windowing system.

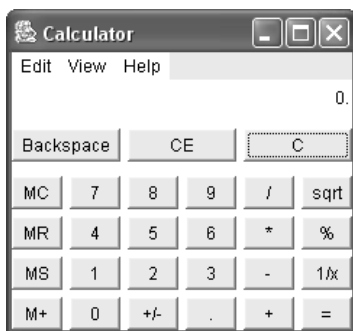
The software library that comes with the J2SDK provides the Java Foundation Classes (JFC) for building GUI-based software applications. Furthermore, JFC provides two sets of packages for building GUIs. They are the Abstract Window Toolkit (AWT) and Swing. Besides AWT and Swing are other windowing-enabling packages developed by other Java communities, such as the Standard Widget Toolkit (SWT).

In the following two subsections, we look at the GUI enabling packages — AWT and Swing.

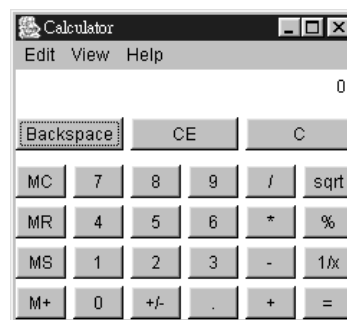
## Abstract Window Toolkit (AWT)

AWT is the first generation of GUI-enabling packages that has come with J2SDK since version 1.0. It provides many GUI widgets (or components) for building GUIs, and its classes interact with the native window system of the machine for building GUIs and handling the interactions between the users and the GUIs.

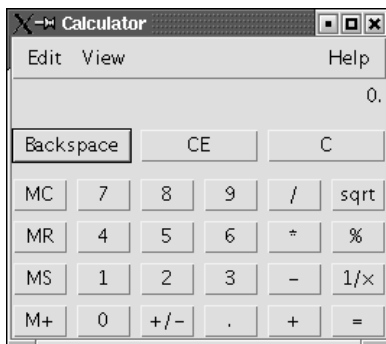
As the GUIs built with AWT classes use native window GUI components to construct the GUI, the appearance of the same GUI-based application varies on different platforms. For illustration, a class, `AWTCalculator`, was written to create a calculator-like GUI. Figure 9.1 shows the screen shots of executing the program on different platforms.



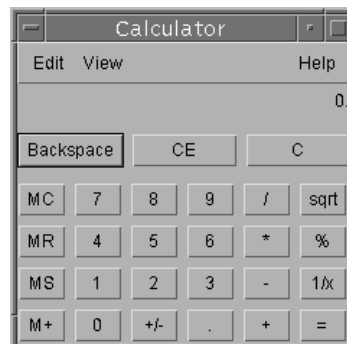
Microsoft Windows XP



Microsoft Windows 9x/NT/2000



KDE/Linux



Sun Solaris

**Figure 9.1** The appearances of the same GUI-based application built with AWT on different platforms

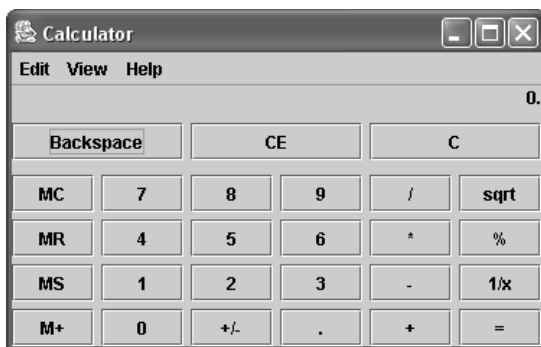
Please refer to the course CD-ROM and website for the definition of the class `AWTCalculator`. By the end of the unit, you will have acquired the necessary knowledge to build the above calculator-like GUI.

Figure 9.1 illustrates that the same GUI application built with AWT can be shown properly on different platforms without any modification, but the appearances during execution, such as background colours and typefaces, are different for different platforms. Building GUIs with AWT fulfils the cross-platform requirement, but it is undesirable that different appearances for the same software application should exist. Even worse is that the functionalities of some GUI components rely on the native window system, and they therefore may behave differently for different platforms.

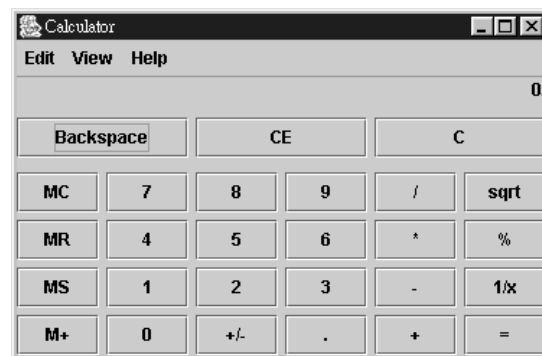
The problems mentioned above stimulated the development of second-generation GUI packages, such as Swing.

## Swing

Swing is also part of JFC. It provides classes for building the next generation of GUI-based software applications. The Swing classes are implemented based on the AWT windowing capability, and the Swing classes themselves handle the appearances of the GUI components instead of relying on the native window system. Therefore, the appearances of the Swing GUI components are the same for all platforms. For example, another calculator-like GUI, the class `SwingCalculator`, was written. The following are the screen shots of the program for different platforms.



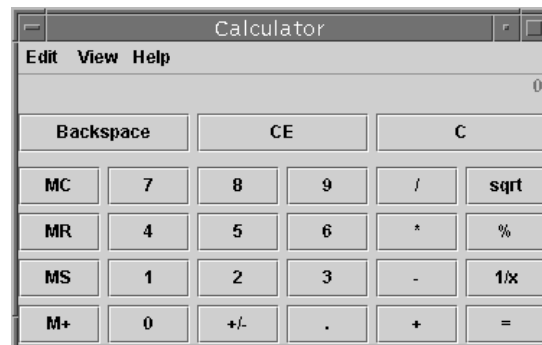
Microsoft Windows XP



Microsoft Windows 9x/NT/2000



KDE/Linux



Sun Solaris

**Figure 9.2** The appearances of the same GUI-based application built with Swing on different platforms

You can see that the decorations for a GUI — including the title bar, control menu and the boundaries — are platform dependent. However, the appearances of the core parts of the GUIs, including the menu bar, buttons and so on, for different platforms are the same. Because the appearances of the Swing GUI components are handled by the Swing components and do not rely on the native window widgets for constructing the GUI, Swing GUI components are considered to be lightweight.

In addition to providing consistent appearances across platforms, Swing provides more GUI components than AWT does. As a result, Swing is currently the preferred package for building Java GUIs. In the following sections in this unit, we concentrate on developing GUIs with Swing components.

## Creating a simple graphical user interface

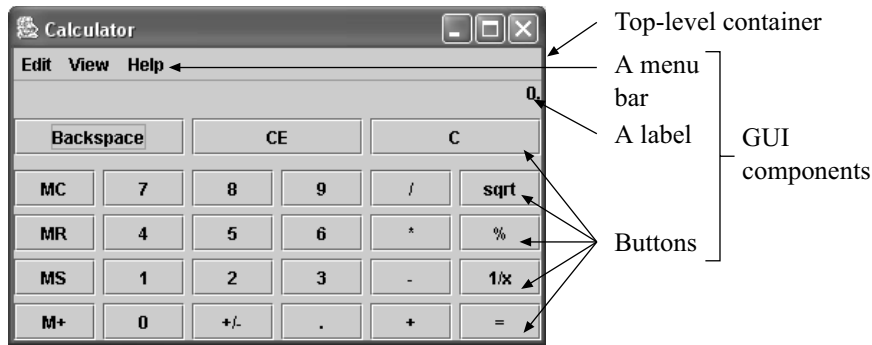
Creating GUIs with the Java programming language usually involves extending and creating objects of the classes provided by the software library. In the following sections, we discuss these GUI components one by one, so that you can enrich your GUI as you learn about more GUI components.

### The steps in creating a graphical user interface

The steps in creating GUIs with the Java programming language and Swing components are as follows:

- 1 *Creating a top-level container object.* Most GUI components, such as a button, label and checkbox, cannot be shown on the screen by themselves. Instead, the Java software library defines top-level container classes so that objects of these classes can be shown on the screen.
- 2 *Creating GUI components and adding them to the top-level container object.* A top-level container object shown on the screen by itself is useless. Therefore, we need to add some GUI components to it so that the top-level container object embeds those GUI components. Whenever the top-level container object is shown on the screen, all embedded GUI component objects are shown on the screen as well, and the user can then operate the GUI.
- 3 *Adding the functionalities to the GUI components.* Even though the GUI components are shown on the screen, such as a button shown in a top-level container object, the GUI performs nothing when you operate the components, such as clicking a button. When you click a button in a GUI, it is an event received by the GUI and it is necessary to teach the GUI how to handle it. That is, you need to define how a GUI responds by writing some methods. Event handling is discussed after we finish discussing the building of GUIs.

A top-level container can be a standalone window shown on the screen. GUI components are the widgets that compose the GUI. For example, the `SwingCalculator` is shown in Figure 9.3 to illustrate the top-level container and GUI components.



**Figure 9.3** Top-level container and GUI components

In Figure 9.3, the GUI is made up of the following components:

- 1 A top-level container that is an independent window on the screen that can embed other GUI components, such as the menu bar, the label containing the number 0 and the calculator buttons.
- 2 A menu bar is a GUI component that consolidates many operations that the application provides and partitions all operations in groups as pull-down menus. If you click a menu on the menu bar, its drop-down menu list, if any, will be shown.
- 3 A label is commonly used to show data, such as text, on the screen but is not editable by the user. In the figure, it is the component showing the number 0 and it occupies the space between the menu bar and the button groups.
- 4 A button is a common GUI component that the user can press to invoke a particular operation of the application. There are 27 buttons in the figure.

## A few GUI components: frame, button and label

### JFrame

As mentioned earlier, the first step in creating a GUI is to create a top-level container object. The software library that comes with the J2SDK provides a class, `javax.swing.JFrame` (or simply `JFrame`) that is a top-level container class. It is commonly used as the top-level container of a GUI-based application.

When a `JFrame` object is created, it is invisible. Before showing it on the screen, other GUI components are added to it so that the `JFrame` object embeds the components. Afterwards, the `JFrame` object can be shown on the screen, and it can be moved and resized like a usual GUI-based application.

To create a `JFrame` object, we need to know its constructor so that we can pass the necessary supplementary data to it. There are four overloaded constructors in total. The following two are usually used:

```
public JFrame()  
public JFrame(String title)
```

A `JFrame` object created by the first constructor has an empty title bar, whereas the reference of a `String` object passed to the second constructor will be used as the title of the frame. When a `JFrame` object is ready to be shown on the screen, we can send a message `show` to it.

We now have sufficient knowledge to create our first GUI. An example program, the `TestJFrame` class, is written as shown in Figure 9.4 to illustrate the steps in creating and showing a `JFrame` object.

```
// Resolve class JFrame  
import javax.swing.*;  
  
// Definition of class TestJFrame1  
public class TestJFrame1 {  
  
    // Main executive method  
    public static void main(String args[]) {  
        // Create a JFrame object with a title  
        JFrame frame = new JFrame("TestJFrame1");  
  
        // Show the JFrame object on the screen  
        frame.show();  
    }  
}
```

**Figure 9.4** TestJFrame1.java

In Figure 9.4, the `import` statement

```
import javax.swing.*;
```

is necessary to resolve the `JFrame` class to be `javax.swing.JFrame`. Alternatively, it is possible to use following `import` statement:

```
import javax.swing.JFrame;
```

However, when the definitions of your classes use more classes in the package `javax.swing`, the former `import` statement is simpler.

In the `main()` method, the following statement is executed:

```
JFrame frame = new JFrame("TestJFrame1");
```

A `JFrame` object is created, and the reference of a `String` with contents "TestJFrame1" is passed to the constructor so that the created `JFrame` object has the title `TestJFrame1`. For the time being, the `JFrame` object is still invisible. As no GUI components are to be added to it, we can

now call its `show()` method to request it to be shown on the screen, that is:

```
frame.show();
```

Then, the following GUI is shown on the screen:



**Figure 9.5** A `JFrame` object without any embedded GUI components

The GUI shown in Figure 9.5 does not show its content by default. It therefore shows the title bar, Minimize, Maximize and Close buttons and the boundary only. Furthermore, because of the limited width, the title "TestJFrame1" is not entirely shown. By resizing the `JFrame` object with your mouse, the object becomes:



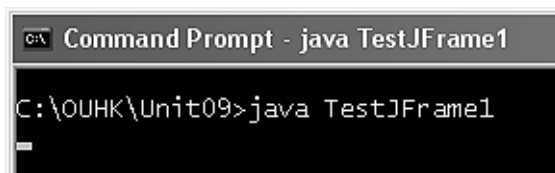
**Figure 9.6** A `JFrame` object after resizing

Now, the entire title of the `JFrame` is shown properly. The content is empty because no GUI components were added to it. If you want to set the size of a `JFrame` object explicitly, you can call its `setSize()` method with supplementary data width and height. For example, the statement

```
frame.setSize(100, 100)
```

sets the size of the `JFrame` object to 100 pixels by 100 pixels.

You can operate the `JFrame` object as if it is a usual window application. For example, you can click the <Minimize> and <Maximize> buttons of the `JFrame` object to minimize and maximize it. Furthermore, clicking the <Close> button removes the `JFrame` object from the screen. However, when you switch to the Command Prompt, you will see that the execution of the program has not been terminated.



**Figure 9.7** The program is still executing after the <Close> button of the `JFrame` object is clicked.

To terminate the `TestJFrame` program explicitly, you have to press the keystroke <Ctrl-C> in the Command Prompt, because the default behaviour of a `JFrame` object when the <Close> button is clicked is to *hide* the `JFrame` object. To change such behaviour of a `JFrame` object, you can call its `setDefaultCloseOperation` with supplementary data of `EXIT_ON_CLOSE`, which is a final class variable of the `JFrame` class. As it is public, we can use it by using the expression, `JFrame.EXIT_ON_CLOSE`. The complete statement is therefore:

```
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

By adding the statement, another testing program, `TestJFrame2.java`, is written as shown in Figure 9.8.

```
// Resolve class JFrame
import javax.swing.*;

// Definition of class TestJFrame2
public class TestJFrame2 {

    // Main executive method
    public static void main(String args[]) {
        // Create a JFrame object with a title
        JFrame frame = new JFrame("TestJFrame2");

        // Set the behaviour of clicking the close button is exit
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Show the JFrame object on the screen
        frame.show();
    }
}
```

**Figure 9.8** `TestJFrame2.java`

You can experiment with executing the `TestJFrame2` program and click the <Close> button of the `JFrame` object to terminate the program by default.

Besides the final class variable `EXIT_ON_CLOSE`, it is possible to call the `setDefaultCloseOperation()` method with other class variables. Their effects are presented in Table 9.1.



**Table 9.1** The effect of calling `setDefaultCloseOperation` with different supplementary data

Final class variable	Effect
<code>DO_NOTHING_ON_CLOSE</code>	Clicking the <Close> button causes nothing to happen. Clicking the <Close> button event is handled by the event handler only.
<code>HIDE_ON_CLOSE</code>	The <code>JFrame</code> object will be hidden. It is the default operation of a <code>JFrame</code> .
<code>DISPOSE_ON_CLOSE</code>	The <code>JFrame</code> object will be hidden; any graphical resources acquired will be released.
<code>EXIT_ON_CLOSE</code>	The method <code>System.exit()</code> is executed, and the program will then terminate.

As the default close operation of a `JFrame` is denoted by `HIDE_ON_CLOSE`, clicking the <Close> button will only hide it by default and the program will not be terminated.

Whenever necessary, it is possible to hide a `JFrame` object programmatically by calling its `hide()` method. Another method `setVisible()` can be used to show or hide it. For example, for a `JFrame` object referred by variable `frame`, the following two statements show and hide it respectively:

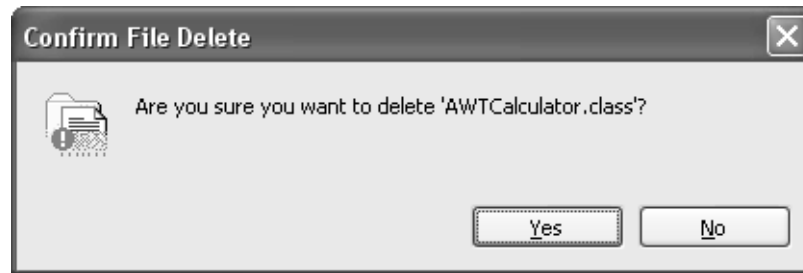
```
frame.setVisible(true); // Show the JFrame object
frame.setVisible(false); // Hide the JFrame object
```

We can now learn how to create a GUI with embedded GUI components, such as the GUI as shown in Figure 9.9 with a button.

**Figure 9.9** A frame with a button

## **JButton**

A button is a common GUI component for getting a response from a user or enabling the user to perform a particular operation. For example, the following is a confirm dialog box with two buttons for getting the user's response.



**Figure 9.10** A confirmation dialog box with two buttons

Another common example of the application of buttons is the button in the toolbar of a window application, such as Microsoft Word as shown in Figure 9.11.



**Figure 9.11** A toolbar of a windows application

In Figure 9.11, the toolbar embeds a list of buttons. Clicking a button performs a particular pre-defined operation.

In Swing, a `JButton` object models a button. Calling the constructor with supplementary data of the reference of a `String` object creates a `JButton` object with the `String` object as its title (or text). For example, the following statement

```
new JButton("Please click me");
```

creates a `JButton` with "Please click me" as the title.

The GUI shown in Figure 9.9 *is a* `JFrame` object and *has a* button with the title "Please click me". It reminds us that 'is a' and 'has a' relationships exist in writing the class for the GUI. If the class of the GUI shown in Figure 9.9 is named `FrameWithButton1`, we have the following:

A `FrameWithButton1` *is a* `JFrame` and a `FrameWithButton1` *has a* `JButton`.

The `FrameWithButton1` is a `JFrame` because we learned that a `FrameWithButton1` has the same appearance and behaviours of a `JFrame` object. As it embeds a `JButton` object, it possesses or has a `JButton`. According to what you learned in *Unit 7*, the 'is a' relationship hints that the class `FrameWithButton1` is a subclass of `JFrame` and the 'has a' relationship suggests that the class `FrameWithButton1` defines a variable of type `JButton` to refer to the `JButton` object. As a result, the template of the class for the above GUI is:

```
public class FrameWithButton1 extends JFrame {
    private JButton button;

    public FrameWithButton1() {
```

```

        .....
        button = new JButton("Please click me");
        .....
    }
}

```

Even though the `JButton` object is created in the constructor of the `FrameWithButton1` class that realizes the ‘has a’ relationship a `FrameWithButton1` has a `JButton`, the `JButton` will not be shown. The reason is that a `JFrame` object has a container object (the content pane) that governs the components to be shown in the `JFrame` window. It is therefore necessary to add the `JButton` object to such a content pane so that whenever the `JFrame` object is shown on the screen, the `JButton` object is shown on the screen as well. To get the content pane object of a `JFrame`, call its `getContentPane()` method. The return type is `java.awt.Container` (or simply `Container` class). That is:

```

Container container = getContentPane();
container.add(button);

```

According to what you have learned so far, we can define the `FrameWithButton1` class as shown in Figure 9.12.

```

// Resolve class Container
import java.awt.*;
// Resolve class JFrame and JButton
import javax.swing.*;

// Definition of class FrameWithButton1
public class FrameWithButton1 extends JFrame {
    // Attribute
    private JButton button;    // The button that the frame possesses

    // Constructor
    public FrameWithButton1() {
        // Call super class constructor with a title
        super("Frame With Button");

        // Create a JButton
        button = new JButton("Please click me");

        // Get the content pane
        Container contentPane = getContentPane();

        // Add the JButton object to the content pane
        contentPane.add(button);

        // Set when the close button is clicked, the application exits
        setDefaultCloseOperation(EXIT_ON_CLOSE);
    }
}

```

**Figure 9.12** `FrameWithButton1.java`

In the constructor of the `FrameWithButton1` class, the first statement is a `super()` statement that provides supplementary data as the reference of a `String` object to the constructor of its immediate superclass, `JFrame`, for the initialization process:

```
super("Frame With Button");
```

As the constructor of the `JFrame` class with a parameter of type `String` is called, the `String` object supplied will be used as the title of the `JFrame` object. Therefore, the statement `super()` with a `String` object will set the title of the `FrameWithButton1` object being created.

Afterwards, the statement

```
button = new JButton("Please click me");
```

creates a `JButton` object with title "Please click me". Then, it is necessary to add the `JButton` object to the `FrameWithButton1` object via its content pane so that it is shown on the screen when the `FrameWithButton1` object is shown. The statements are:

```
Container contentPane = getContentPane();  
contentPane.add(button);
```

Finally, to make the application terminate when the <Close> button is clicked, the following statement is executed as well in the constructor:

```
setDefaultCloseOperation(EXIT_ON_CLOSE);
```

Please note that the ways the `setDefaultCloseOperation()` method are called in the `TestJFrame2` class `main()` method and the `FrameWithButton` constructors are different. In the `TestJFrame2` class `main()` method, the method call is:

```
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

The method call in the `FrameWithButton1` constructor is:

```
setDefaultCloseOperation(EXIT_ON_CLOSE);
```

For the `TestJFrame2` class, the `JFrame` object is referred by the variable `frame` and the message `setDefaultCloseOperation` is sent to the `JFrame` object via the variable `frame`. The `FrameWithButton1` class is a subclass of `JFrame` and therefore inherits all methods and attributes from the `JFrame` class, including the `setDefaultCloseOperation()` method and the class variable `EXIT_ON_CLOSE`. Therefore, the statement is equivalent to:

```
this.setDefaultCloseOperation(this.EXIT_ON_CLOSE);
```

The interpretation is that the supplementary data of its class variable `EXIT_ON_CLOSE` are sent with the message `setDefaultCloseOperation` to itself.

To test the `FrameWithButton1` class, we need a driver program that is similar to the `TestJFrame2` class. A class `TestFrameWithButton1` is written as shown in Figure 9.13.

```
// Definition of class TestFrameWithButton1
public class TestFrameWithButton1 {

    // Main executive method
    public static void main(String args[]) {
        // Create a FrameWithButton1 object
        FrameWithButton1 frame = new FrameWithButton1();

        // Show it on the screen
        frame.show();
    }
}
```

**Figure 9.13** `TestFrameWithButton1.java`

Compile the classes and execute the `TestFrameWithButton1`. A GUI as shown in Figure 9.14 is shown on the screen.



**Figure 9.14** A `FrameWithButton1` object

The GUI shown is similar to the `JFrame` object created by the `TestJFrame2`, but it actually has a `JButton` object. To show the `JButton`, resize the object manually. The GUI will become as shown in Figure 9.15.



**Figure 9.15** A `FrameWithButton1` object after resizing

No matter how you resize the GUI, the `JButton` always occupies the entire contents. If the size of the GUI is not large enough, the button title cannot be completely shown as shown in Figure 9.16.



**Figure 9.16** A `FrameWithButton1` object that is resized improperly

It is tedious to resize the GUI manually, and it is hard to tell the best size so that all embedded components are all properly shown and the size is optimal. Fortunately, the `JFrame` class defines a `pack()` method that

can set all embedded GUI components to their optimal or preferable size and the size of the JFrame is set to be optimal.

The `FrameWithButton1` is modified as `FrameWithButton2` by adding the statement to call the `pack()` method. Its definition is shown in Figure 9.17.

```
// Resolve class Container
import java.awt.*;

// Resolve class JFrame and JButton
import javax.swing.*;

// Definition of class FrameWithButton2
public class FrameWithButton2 extends JFrame {
    // Attribute
    private JButton button;    // The button that the frame possesses

    // Constructor
    public FrameWithButton2() {
        // Call super class constructor with a title
        super("Frame With Button");

        // Create a JButton
        button = new JButton("Please click me");

        // Get the content pane
        Container contentPane = getContentPane();

        // Add the JButton object to the content pane
        contentPane.add(button);

        // Set when the close button is clicked, the application exits
        setDefaultCloseOperation(EXIT_ON_CLOSE);

        // Reorganize the embedded components
        pack();
    }
}
```

**Figure 9.17** `FrameWithButton2.java`

A driver program, `TestFrameWithButton2`, is written to test the `FrameWithButton2` class. As it is almost exactly the same as the `TestFrameWithButton1` except the object to be created is a `FrameWithButton2` object, the definition is not shown here. Please refer to the course CD-ROM and website.

Compile the classes `FrameWithButton2` and `TestFrameWithButton2`. The GUI as shown in Figure 9.18 is shown on the screen:



**Figure 9.18** A `FrameWithButton2` object

The size of the `JButton` object of the `FrameWithButton2` is optimal; so is that of the `FrameWithButton2` object.

The text on a `JButton` is not only set by passing the reference of a `String` object to its constructor, but it is also set by calling its `setText()` method any time after the `JButton` is created or even is shown on the screen. Therefore, according to the requirements of your application, it is possible to programmatically change the text on a `JButton`.

The use of the `setText()` method is illustrated in the `FrameWithButton3` class as shown in Figure 9.19.

```
// Resolve class Container
import java.awt.*;
// Resolve class JFrame and JButton
import javax.swing.*;

// Definition of class FrameWithButton3
public class FrameWithButton3 extends JFrame {
    // Attribute
    private JButton button;    // The button that the frame possesses

    // Constructor
    public FrameWithButton3(String frameTitle, String buttonText) {
        // Call super class constructor with a title
        super(frameTitle);

        // Create a JButton
        button = new JButton();
        button.setText(buttonText);

        // Get the content pane
        Container contentPane = getContentPane();

        // Add the JButton object to the content pane
        contentPane.add(button);

        // Set when the close button is clicked, the application exits
        setDefaultCloseOperation(EXIT_ON_CLOSE);

        // Reorganize the embedded components
        pack();
    }
}
```

**Figure 9.19** `FrameWithButton3.java`

As well as using the `setText()` method for setting the text of the `JButton` object, the `FrameWithButton3` class defines the constructor with a parameter list of two parameters of the type `String`. They specify the title of the `JFrame` object and the text of the `JButton` respectively. Accordingly, the statement for creating the `FrameWithButton3` object in the `TestFrameWithButton3` class `main()` method has to be modified as well. The definition is shown in Figure 9.20.

```
// Definition of class TestFrameWithButton3
public class TestFrameWithButton3 {

    // Main executive method
    public static void main(String args[]) {
        // Create a FrameWithButton3 object with the specified
        // frame title and button title
        FrameWithButton3 frame = new FrameWithButton3(
            "Frame With Button", "Please click me");

        // Show the frame on the screen
        frame.show();
    }
}
```

**Figure 9.20** `TestFrameWithButton3.java`

Compared with the `FrameWithButton2` class, the `FrameWithButton3` class is more flexible, because the window title and the button title are supplied to the constructor. Therefore, it is possible to create a `FrameWithButton3` object with a different window title and button title, whereas the window title and button text of a `FrameWithButton2` object is fixed.

## JLabel

On a GUI, there is often some static text that has no operations associated with them, for example the text “Address” on the Microsoft Internet Explorer.



**Figure 9.21** A snapshot of Microsoft Internet Explorer

A `JLabel` object in the `Swing` package models a label. The way to create a `JLabel` is similar to that of a `JButton`. The two common constructors of the `JLabel` class are:

```
public JLabel();
public JLabel(String text);
```



The JLabel object created by the first constructor has no title, but the String object supplied to the latter is used as the title of the created JLabel object.

Once a JLabel object is created, it is possible to add an object of JFrame or its subclass to the top-level container object similar to the scenario for a JButton object previously discussed. Then, we can write a class FrameWithLabel1 that defines the object that is a JFrame and has a JLabel. The definition is similar to the FrameWithButton2 class and is shown in Figure 9.22.

```
// Resolve class Container
import java.awt.*;
// Resolve class JFrame and JLabel
import javax.swing.*;

// Definition of class FrameWithLabel1
public class FrameWithLabel1 extends JFrame {
    // Attribute
    private JLabel label;    // The label that the frame possesses

    // Constructor
    public FrameWithLabel1() {
        // Call super class constructor with a title
        super("Frame With label");

        // Create a JLabel
        label = new JLabel("Hello World");

        // Get the content pane
        Container contentPane = getContentPane();

        // Add the JLabel object to the content pane
        contentPane.add(label);

        // Set when the close button is clicked, the application exits
        setDefaultCloseOperation(EXIT_ON_CLOSE);

        // Reorganize the embedded components
        pack();
    }
}
```

**Figure 9.22** FrameWithLabel1.java

A driver program, TestFrameWithLabel1, is written to test the FrameWithLabel1 class. As it is similar to the previous TestFrameWithButton2 class, it is not shown here.

Compile the classes and execute the TestFrameWithLabel1 class. The following GUI is shown on the screen:



**Figure 9.23** A `FrameWithLabel1` object

Another possible use of labels is to show operation results by updating the text on a label. Based on the class `FrameWithLabel1`, a class `FrameWithLabel2` is written as an illustration. You can find the complete discussion on the class in Appendix A.

Please use the following self-test to experiment with developing a GUI using the Java programming language.

### *Self-test 9.1*

Modify the `FrameWithLabel1` so that the window title and button title are supplied to the `FrameWithLabel` object via a constructor. That is, the constructor of the `FrameWithLabel1` class is modified to be:

```
public FrameWithLabel(String frameTitle, String labelTitle) {  
    ...  
}
```

Furthermore, modify the `TestFrameWithLabel1` so that the first and second program parameters (`args[0]` and `args[1]`) are used as the frame title and label title respectively.

---

## Arranging GUI components with layout managers

In the previous section, you learned how to create GUIs. Two common GUI components — button (`JButton`) and label (`JLabel`) — were introduced as well. If you resize the window, the embedded button or label will be resized to occupy the entire content.

However, a window with either a button or a label is not that useful, and we usually need several GUI components to be embedded in a single GUI. In the sample programs we discussed in the previous section, the method `add()` of the content pane (of type `Container`) is called to add a GUI component to the top-level container. Would two or more GUI components be shown on the GUI if they are added to the content pane by calling the `add()` method one by one?

To answer the question, a class `FrameWithMultiButtons1` is written in Figure 9.24.

```
// Resolve class Container
import java.awt.*;
// Resolve class JFrame and JButton
import javax.swing.*;

// Definition of class FrameWithMultiButtons1
public class FrameWithMultiButtons1 extends JFrame {
    // Attribute
    private JButton button1;    // The first button
    private JButton button2;    // The second button

    // Constructor
    public FrameWithMultiButtons1() {
        // Call super class constructor with a title
        super("Frame With Multiple Buttons");

        // Create a JButton
        button1 = new JButton("Button One");
        button2 = new JButton("Button Two");

        // Get the content pane
        Container contentPane = getContentPane();

        // Add the JButton objects to the content pane
        contentPane.add(button1);
        contentPane.add(button2);

        // Set when the close button is clicked, the application exits
        setDefaultCloseOperation(EXIT_ON_CLOSE);

        // Reorganize the embedded components
        pack();
    }
}
```

**Figure 9.24** `FrameWithMultiButtons1.java`

A driver program, the `TestFrameWithMultiButtons1` class, is written to test the `FrameWithMultiButtons1` class. (Refer to the CD-ROM for the definition of the class `TestFrameWithMultiButtons1`.) Compile the classes and execute the `TestFrameWithMultiButtons1` class. The following GUI is shown on the screen:



**Figure 9.25** The `FrameWithMultiButtons1` object shown on the screen

The GUI shown in Figure 9.25 illustrates that only the last `JButton` object added to the content pane is shown on the screen.

The above illustrates two characteristics of a GUI developed with the Java programming language and Swing:

- 1 The position and size of each GUI component embedded in a GUI are determined and updated automatically at runtime.
- 2 Some rules govern the placing of GUI components on a GUI.

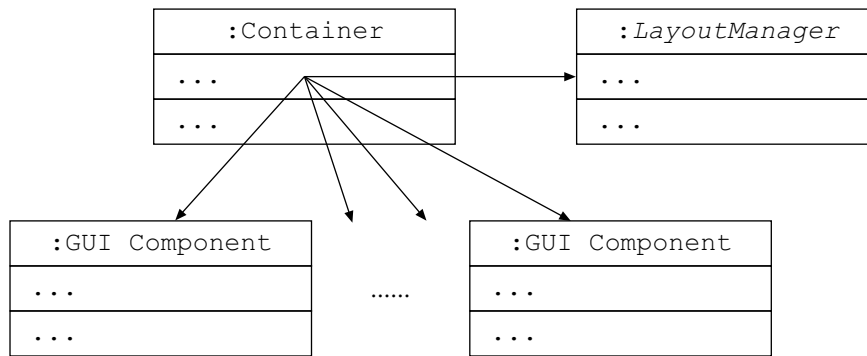
These two characteristics enable a GUI built with the Java programming language and Swing to be cross-platform. The reason is that each container object has a layout manager object to determine the positions and sizes of all embedded GUI components.

## What are layout managers?

You learned that a GUI component has to be added to a top-level container so that it can be shown on the screen. More exactly, it can be a content pane of a `JFrame` object. As we have mentioned, each `JFrame` object has a content pane object or `Container` object. To arrange the GUI components added to a `Container` object, each `Container` has a layout manager object for arranging the GUI components added to the `Container` object. The type of layout manager is `java.awt.LayoutManager` (or simply `LayoutManager`). Whenever a `Container` object is created, such as the content pane of a `JFrame`, a default layout manager is created as well.

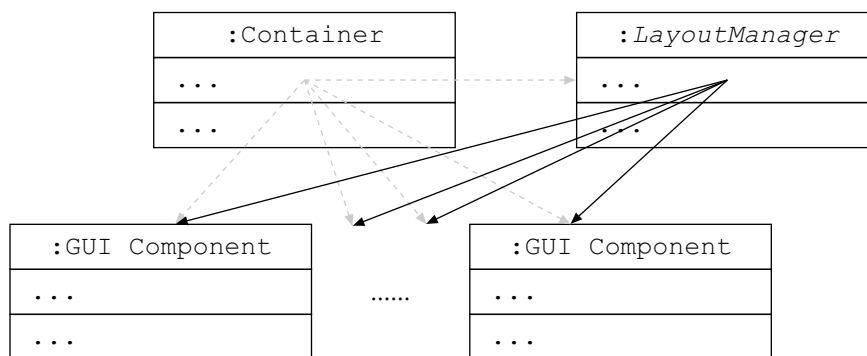
The type `LayoutManager` is an abstract type (and more exactly an interface, discussed in a later section), and it is therefore not possible to create a `LayoutManager` object. Nonetheless, it is possible to create objects of the concrete subclasses of the abstract type `LayoutManager`. Therefore, the `LayoutManager` object to be created by a `Container` object is an object of the concrete subclass of the abstract superclass `LayoutManager`.

The relationships among the `Container`, `LayoutManager` and GUI component objects are visualized in Figure 9.26.



**Figure 9.26** The relationships among `Container`, `LayoutManager` and GUI component objects

In Figure 9.26, the GUI component objects can be a `JButton` object, a `JLabel` object or even other container objects. The arrows in the figure indicate that the `Container` object maintains the references of all embedded GUI components. When a GUI component is added to the `Container` object, the `Container` will pass the reference of the GUI component to the `LayoutManager` object it possesses. Therefore, the scenario of the objects is more exactly like what Figure 9.27 shows.



**Figure 9.27** The `LayoutManager` object of a `Container` object has the references to the GUI component objects added to the `Container` objects

(The arrows in Figure 9.27 are set as dimmed and dashed to highlight the arrows pointing from the `LayoutManager` object to the GUI component objects.)

The `LayoutManager` object of the `Container` object determines the sizes and positions of the GUI components to be shown in the `Container` on the screen. Therefore, it is usual practice to set a proper layout manager object to a `Container` object before adding any GUI component to it.

The Java software library provides different layout manager objects to allow for different arrangements of the GUI components. The three commonly used layout manager classes are `java.awt.FlowLayout` (or `FlowLayout`), `java.awt.BorderLayout` (or `BorderLayout`), and `java.awt.GridLayout` (or `GridLayout`) respectively. We discuss these three layout managers one by one in the following subsections.

## The `FlowLayout` layout manager

The `FlowLayout` layout manager is the simplest layout manager. An arbitrary number of GUI components can be added to the `Container` object with a `FlowLayout` object as its layout manager. They are arranged from left to right in the `Container`. Furthermore, we mentioned that each GUI component has a preferred size, and each GUI component is shown in the `Container` at the preferred size.

To illustrate the `FlowLayout` layout manager, the `FrameWithMultiButtons1` is modified to be `FrameWithFlowLayout` so that each `FrameWithFlowLayout` object has three `JButton` objects with different titles. They are added to the `Container` — the content pane of the `JFrame` — one by one. Please bear in mind that it is necessary to set a `FlowLayout` object in the container before adding any GUI component to the `Container` object. To change the layout manager object of a `Container` object, call the `setLayout()` method of the `Container` object with the reference to a layout manager object.

The definition of the class `FrameWithFlowLayout` is shown in Figure 9.28.

```
// Resolve class Container, FlowLayout
import java.awt.*;
// Resolve class JFrame and JButton
import javax.swing.*;

// Definition of class FrameWithFlowLayout
public class FrameWithFlowLayout extends JFrame {
    // Attribute
    private JButton button1;    // The first button
    private JButton button2;    // The second button
    private JButton button3;    // The third button

    // Constructor
    public FrameWithFlowLayout() {
        // Call super class constructor with a title
        super("Frame With Multiple Buttons");

        // Create a JButton
        button1 = new JButton("Button One");
        button2 = new JButton("Button Two");
        button3 = new JButton("Button Three");

        // Get the content pane
        Container contentPane = getContentPane();

        // Set the layout manager of the content pane to be FlowLayout
        FlowLayout layout = new FlowLayout();
        contentPane.setLayout(layout);

        // Add the JButton objects to the content pane
        contentPane.add(button1);
        contentPane.add(button2);
        contentPane.add(button3);
    }
}
```

```

        // Set when the close button is clicked, the application exits
        setDefaultCloseOperation(EXIT_ON_CLOSE);

        // Reorganize the embedded components
        pack();
    }
}

```

**Figure 9.28** `FrameWithFlowLayout.java`

We need a driver program, say a class `TestFrameWithFlowLayout`, to test the `FrameWithFlowLayout` class. It is similar to the previous driver program — please find it in the course CD-ROM and website.

Compile the classes `FrameWithFlowLayout` and `TestFrameWithFlowLayout` classes. Execute the `TestFrameWithFlowLayout` program. The following GUI as shown in Figure 9.29 is shown on the screen.



**Figure 9.29** The `FrameWithFlowLayout` object

With a `FlowLayout` layout manager, all embedded GUI components are arranged from left to right. Each GUI component is shown with the optimal or preferable size.

## The `BorderLayout` layout manager

Another common layout manager is the `BorderLayout` (`java.awt.BorderLayout`) layout manager. It is the default layout manager of the content pane of a `JFrame` object, which means that a `BorderLayout` object is created explicitly by a `JFrame` object and is used as the layout manager.

To make its behaviours more explicit, a GUI with a `BorderLayout` object as layout manager is shown in Figure 9.30. We will discuss how to create it very soon.



**Figure 9.30** The `FrameWithBorderLayout` object that uses `BorderLayout` as its layout manager

The first observation of a `Container` object with a `BorderLayout` object as its layout manager is that the `Container` is divided into five regions. These regions are referred to as east, south, west, north, and center. Each region can hold one (or no) GUI component. Such a `Container` can therefore embed a maximum of five GUI components. If more than one GUI component is added to the same region, only the last one added to the region is shown.

As each region of a `Container` object with `BorderLayout` as its layout manager is optional, the way to add a GUI component to the `Container` object is different. It is necessary to call the overloaded `add()` method of the `Container` with two parameters — the first is the reference to a GUI component and the second specifies in which region the GUI component is placed. `public` class variables are defined in the `BorderLayout` class for specifying the regions, which are `EAST`, `SOUTH`, `WEST`, `NORTH` and `CENTER` respectively. For example, the statement to add a GUI component object referred by the variable `comp` to the east region of a `Container` object referred by the variable `contentPane` is:

```
contentPane.add(comp, BorderLayout.EAST);
```

If the second parameter is missing, the GUI component is by default added to the center region of the `Container` object.

You can imagine that the way to create the GUI shown in Figure 9.30 is to create five `JButton` objects and add them to the `Container` using statements that look like the one above. The class `FrameWithBorderLayout` shown in Figure 9.31 creates the GUI shown above.

```
// Resolve class Container
import java.awt.*;
// Resolve class JFrame and JButton
import javax.swing.*;

// Definition of class FrameWithBorderLayout
public class FrameWithBorderLayout extends JFrame {
    // Attribute
    private JButton buttonEast;    // The east button
    private JButton buttonSouth;   // The south button
    private JButton buttonWest;    // The west button
    private JButton buttonNorth;   // The north button
    private JButton buttonCenter;  // The center button

    // Constructor
    public FrameWithBorderLayout() {
        // Call super class constructor with a title
        super("Frame With Multiple Buttons");
    }
}
```



```

        // Create JButton objects
        buttonEast = new JButton("East");
        buttonSouth = new JButton("South");
        buttonWest = new JButton("West");
        buttonNorth = new JButton("North");
        buttonCenter = new JButton("Center");

        // Get the content pane
        Container contentPane = getContentPane();

        // Add the JButton objects to the content pane
        contentPane.add(buttonEast, BorderLayout.EAST);
        contentPane.add(buttonSouth, BorderLayout.SOUTH);
        contentPane.add(buttonWest, BorderLayout.WEST);
        contentPane.add(buttonNorth, BorderLayout.NORTH);
        contentPane.add(buttonCenter, BorderLayout.CENTER);

        // Set when the close button is clicked, the application exits
        setDefaultCloseOperation(EXIT_ON_CLOSE);

        // Reorganize the embedded components
        pack();
    }
}

```

**Figure 9.31** FrameWithBorderLayout.java

You should note that it is not necessary to set the layout manager of the GUI in the constructor, as the content pane of a `JFrame` object uses a `BorderLayout` object by default.

The driver program, `TestFrameWithBorderLayout`, can be found in the course CD-ROM and website. Execute it and it will show the GUI shown in Figure 9.30.

## The GridLayout layout manager

In Figure 9.1 and Figure 9.2, the digit buttons on the calculator-like GUI are arranged in rows and columns. Such a style of GUI component arrangement is achievable by using the `java.awt.GridLayout` (`GridLayout`) layout manager.

Before you can set a `GridLayout` object to be the layout manager of a `Container`, you have to know how to create a `GridLayout` object. The `GridLayout` class defines overloaded constructors. The most common one is:

```
public GridLayout(int rows, int cols)
```

The two parameters of type `int` specify the numbers of rows and columns of the regions partitioned in the space provided by the `Container` object. Then, GUI components are added to the `Container` object from left to right, top to bottom. For example, the GUI shown in

Figure 9.32 has six JButton objects on it. They are arranged in three rows and two columns.



**Figure 9.32** A `FrameWithGridLayout` object

To create such a GUI, it is necessary to create a `GridLayout` object that arranges objects in three rows and two columns. The statement to create such a `GridLayout` object is:

```
new GridLayout(3, 2)
```

Afterwards, it is set to the `Container` object. The GUI components can then be added to it. Suppose that the `Container` object is referred by variable `contentPane`. The program segment is:

```
contentPane.setLayout(new GridLayout(3, 2));
contentPane.add(button1);
...
contentPane.add(button6);
```

The complete definition of the class `FrameWithGridLayout` for the above GUI is shown in Figure 9.33.

```
// Resolve class Container, GridLayout
import java.awt.*;
// Resolve class JFrame and JButton
import javax.swing.*;

// Definition of class FrameWithGridLayout
public class FrameWithGridLayout extends JFrame {
    // Attribute
    private JButton button1;    // The first button
    private JButton button2;    // The second button
    private JButton button3;    // The third button
    private JButton button4;    // The fourth button
    private JButton button5;    // The fifth button
    private JButton button6;    // The sixth button

    // Constructor
    public FrameWithGridLayout() {
        // Call super class constructor with a title
        super("Frame With Multiple Buttons");
```

```

// Create JButton objects
button1 = new JButton("One");
button2 = new JButton("Two");
button3 = new JButton("Three");
button4 = new JButton("Four");
button5 = new JButton("Five");
button6 = new JButton("Six");

// Get the content pane
Container contentPane = getContentPane();

// Set the layout manager of the content pane to be GridLayout
contentPane.setLayout(new GridLayout(3, 2));

// Add the JButton objects to the content pane
contentPane.add(button1);
contentPane.add(button2);
contentPane.add(button3);
contentPane.add(button4);
contentPane.add(button5);
contentPane.add(button6);

// Set when the close button is clicked, the application exits
setDefaultCloseOperation(EXIT_ON_CLOSE);

// Reorganize the embedded components
pack();
}

```

**Figure 9.33** FrameWithGridLayout.java

A driver program, `TestFrameWithGridLayout`, is written. It is available on the course CD-ROM and website. Compile the classes and execute the `TestFrameWithGridLayout`. The GUI in Figure 9.32 is shown.

If the statement for creating the `GridLayout` object in the definition of class `FrameWithGridLayout` were changed to

```
contentPane.setLayout(new GridLayout(2, 3));
```

the GUI created would have been:



**Figure 9.34** A `FrameWithGridLayout` with two rows and three columns

The GUIs shown in Figure 9.33 and Figure 9.34 indicate that all the embedded GUI components are the same size. That is, their widths and heights are the same. Therefore, all GUI components are not shown in

their own preferred size. More exactly, the common dimensions of the GUI components are determined to be the maximum preferred width and maximum preferred height among all components.

We have discussed three common layout manager classes so that you can visualize the arrangements of the GUI components with different layout manager objects. As the size of the container can be changed, the arrangements of the GUI components after the container is resized are different for different layout managers. Appendix B provides some insight into the behaviours of different layout managers after resizing the containers.

### Self-test 9.2

- 1 Write a class that prepares the following GUI with three buttons:



What layout manager(s) can be used to build the above GUI? If the GUI can be built with more than one layout manager, what are the differences?

- 2 Write a class that prepares the following keypad-like GUI:



Up to this point, we have discussed two common GUI components — `JButton` and `JLabel` — for modelling a GUI button and label respectively. GUIs usually need more GUI component types, such as text-fields, checkboxes, radio-boxes and menu bars. We discuss in the following section how to use them.

## Other GUI components

In the two common GUI components we have discussed so far — `JLabel` and `JButton` — the `JLabel` class models labels showing static text on the GUI, and the `JButton` class models buttons on a GUI for the users to trigger some operations. Besides these two GUI components, we usually need some entry fields for the users to enter textual data for processing. The following subsection discusses other GUI components that are commonly used for constructing GUIs.

### Text fields

Text fields and text areas are common GUI components that enable users to enter textual data for further processing. A difference between them is that a text field is a single line of textual entry, whereas a text area enables the user to enter a block of text.

In Swing, a text field is modelled by `javax.swing.JTextField` (or simply `JTextField`). It defines several constructors for creating `JTextField` objects. The following two are most commonly used:

```
public JTextField(int columns)
public JTextField(String text, int columns)
```

For the first constructor, the supplementary data supplied determines the width of the text field according to number of columns. The second constructor accepts one more supplementary data item as the initial content of the text field.

To test the `JTextField` class, a class `FrameWithTextField` is written in Figure 9.35.

```
// Resolve classes in javax.swing package
import java.awt.*;
import javax.swing.*;

// Definition of class FrameWithTextField
public class FrameWithTextField extends JFrame {
    // Attribute
    private JTextField textfield = new JTextField(10);

    public FrameWithTextField() {
        super("Frame With TextField");

        // Get the content pane object
        Container contentPane = getContentPane();

        // Set the layout manager to be a FlowLayout object
        contentPane.setLayout(new FlowLayout());

        // Add the JTextField object to the content pane
        contentPane.add(textfield);
    }
}
```

```

        // Set the default operation with the close button is clicked
        setDefaultCloseOperation(EXIT_ON_CLOSE);

        // Set the GUI to its optimal size
        pack();
    }
}

```

**Figure 9.35** FrameWithTextField.java

A driver program `TestFrameWithTextField` is written and can be found on the course CD-ROM and website. Compile the classes and execute the `TestFrameWithTextField`. The frame is shown in Figure 9.36.

**Figure 9.36** The GUI modelled by a `FrameWithTextField` object is shown on the screen

You should notice that the layout manager of the content pane is set to be a `FlowLayout` object. Then, when the window is resized, the size of the text field is kept unchanged.

Even though the number of columns of the text field is set to be 10, it does not limit the number of characters you can type into the text field. If you type more characters than the text field can accommodate, the characters in the text field will scroll to the left. Furthermore, the width of the text field is set to be the width of 10 times the average character width. Therefore, more than or fewer than 10 characters may occupy the entire text field, as shown in Figure 9.37.

**Figure 9.37** Different numbers of characters may occupy the entire text field

## Text areas

The Java standard software library provides the `JTextArea` object to model a text area that provides multiple line textual data entry to the user. It defines several constructors. You usually use either one of the following two constructors to create a `JTextArea` object:

```

public JTextArea(int rows, int columns)
public JTextArea(String text, int rows, int columns)

```

The parameters `rows` and `columns` supplied to these two constructors specify the number of visible rows and visible columns in the text area. If you want to provide an initial content to the text area, you can create the `JTextArea` object by the second constructor and provide a `String` object as its initial content.

A sample class `FrameWithTextArea1` written to illustrate the use of the `JTextArea` class is shown in Figure 9.38.

```
// Resolve classes in javax.swing package
import javax.swing.*;

// Definition of class FrameWithTextArea1
public class FrameWithTextArea1 extends JFrame {
    // Attribute
    private JTextArea textArea;

    public FrameWithTextArea1() {
        // Set the frame title
        super("Frame With TextArea");

        // Construct the contents of the text area
        String content = "Hello World!\n";
        for (int i=0; i < 2; i++) {
            content += content;
        }

        // Create a JTextArea object
        textArea = new JTextArea(content, 5, 40);

        // Add the text area to the content pane of the frame
        getContentPane().add(textArea);

        // Set the default operation if the close button of the frame
        // is clicked
        setDefaultCloseOperation(EXIT_ON_CLOSE);

        // Set the GUI to its optimal size
        pack();
    }
}
```

**Figure 9.38** `FrameWithTextArea1.java`

The corresponding driver program is the `TextFrameWithTextArea1` class, which can be found on the course CD-ROM and website. Compile the classes and execute the `TextFrameWithTextArea1` program. A window as shown in Figure 9.39 will appear on the screen.



**Figure 9.39** The window shown by executing the `TextFrameWithTextArea1` program

Before creating the `JTextArea` object while executing the `FrameWithTextArea1` constructor, a `String` object is prepared containing four lines of "Hello World" that are separated by the new-line character (`\n`). You can now edit the contents of the text area by adding new text to it or removing some of it.

If you add new lines to the text area, you should notice that those lines cannot be shown, but it is expected that a scroll bar would appear to allow you to scroll through the entire text area. You can implement such a behaviour by using a `JScrollPane` object to contain the `JTextArea` object. It is added to the content pane of the frame instead. The class `FrameWithTextArea2` written in Figure 9.40 uses a `JScrollPane` object.

```
// Resolve classes in javax.swing package
import javax.swing.*;

// Definition of class FrameWithTextArea2
public class FrameWithTextArea2 extends JFrame {
    // Attribute
    private JTextArea textArea;

    public FrameWithTextArea2() {
        // Set the frame title
        super("Frame With TextArea");

        // Construct the contents of the text area
        String content = "Hello World!\n";
        for (int i=0; i < 3; i++) {
            content += content;
        }

        // Create a JTextArea object
        textArea = new JTextArea(content, 5, 40);

        // Create a JScrollPane object
        JScrollPane scrollPane = new JScrollPane(textArea);

        // Add the text area to the content pane of the frame
        getContentPane().add(scrollPane);

        // Set the default operation if the close button of the frame
        // is clicked
        setDefaultCloseOperation(EXIT_ON_CLOSE);

        // Set the GUI to its optimal size
        pack();
    }
}
```

**Figure 9.40** `FrameWithTextArea2.java`

Compile the `FrameWithTextArea2` class with the driver program `TestFrameWithTextArea2` that can be found on the course CD-ROM



and website. Execute the `TextFrameWithTextArea2` program. A window like Figure 9.41 is shown on the screen.



**Figure 9.41** A text area with scroll bar(s) by using a `JScrollPane` object to contain a `JTextArea` object

By using a `JScrollPane` object, the vertical scrollbar and horizontal scrollbar appear whenever necessary. Therefore, if you remove some lines from the text area, the vertical scroll bar will be hidden. However, if you enter characters in a single line that exceed the visible region of a `JTextArea`, the horizontal scrollbar will appear as shown in Figure 9.42.



**Figure 9.42** Both vertical scrollbar and horizontal scrollbars are shown whenever necessary

## Accessing text fields and text areas

The contents of a text field or text area can be retrieved or assigned any time during the program execution by using the `getText()` and `setText()` methods of a `JTextField` object or a `JTextArea` object. Later in the unit, we discuss how these two methods are frequently used to handle user interactions programmatically.

As both `JTextField` and `JTextArea` are subclasses of the class `JTextComponent`, the two subclasses inherit the same set of methods from the superclass. A commonly used inherited method is `setEditable()`. Both a `JTextField` object and a `JTextArea` object by default enable the user to modify the contents. However, if the `setEditable()` method of such an object is called with supplementary data of `false`, the contents of the component become non-editable by the user but can still be modified by calling their `setText()` method.

If a text field becomes non-editable, it behaves like a textual label. A non-editable text area is usually used to show a long user agreement so that the user can read the contents but cannot edit them.

## Canvas

A canvas is a GUI component that models a drawing board on a GUI so that you can draw anything on it arbitrarily. In the Java standard software library, the Swing packages do not provide a Swing version of canvas, but AWT does. The class is `java.awt.Canvas` (or simply `Canvas` class). A common way to create a `Canvas` object is by using the following constructor:

```
public Canvas()
```

This means that no supplementary data are required. The preferable size of a `Canvas` object is zero by zero. Unless the size of a `Canvas` object is set explicitly by calling its `setSize()` method or it is placed in a container with a layout manager that adjusts the size of the `Canvas` object, a `Canvas` object is by default invisible.

The `Canvas` class has a `paint()` method that is called to construct its appearance whenever it is shown on the screen for the first time or is uncovered. The declaration of the `paint()` method is:

```
public void paint(Graphics g)
```

The fully qualified name of the `Graphics` class is `java.awt.Graphics` and it defines many methods for drawing and painting.

At runtime, each `Canvas` object is associated with a `Graphics` object. The effects of calling the drawing and painting methods of the `Graphics` object will appear within the region occupied by the `Canvas` object on the screen. For example, when the `paint()` method is called, the `Graphics` object associated with the `Canvas` object is supplied to the method, so that the `paint()` method can use the `Graphics` object to construct its appearance. For a detailed description of the usage of the `Graphics` class, please refer to its API documentation.

To have your own canvas, a usual practice is to write a subclass of the `Canvas` class and define your own `paint()` method. A simple example, `HelloCanvas`, is written as shown in Figure 9.43 that shows the message “Hello World” within it.

```
// Resolve classes in java.awt package
import java.awt.*;

// Definition of class HelloCanvas
public class HelloCanvas extends Canvas {

    // Override the paint method so that when the Canvas is shown on
    // the screen, the message "Hello World" is shown within the canvas
    public void paint(Graphics g) {
        // Show the message
        g.drawString("Hello World", 20, 20);
    }
}
```

Figure 9.43 `HelloCanvas.java`

It is now possible to create a `HelloCanvas` object and add it to a `JFrame` as a usual GUI component. A sample program, the `FrameWithHelloCanvas` class (Figure 9.44) illustrates how to do so.

```
// Resolve classes in java.awt and javax.swing packages
import java.awt.*;
import javax.swing.*;

// Definition of class FrameWithHelloCanvas
public class FrameWithHelloCanvas extends JFrame {
    // The drawing board to be shown on the GUI
    private Canvas myCanvas = new HelloCanvas();

    // Constructor
    public FrameWithHelloCanvas() {
        // Set frame title
        super("Frame with HelloCanvas");

        // Get the content pane
        Container contentPane = getContentPane();

        // Add the canvas object to content pane
        contentPane.add(myCanvas, BorderLayout.CENTER);

        // Set the size of the canvas explicitly
        myCanvas.setSize(100, 50);

        // Set the default operation if the close button of the frame
        // is clicked
        setDefaultCloseOperation(EXIT_ON_CLOSE);

        // Set the frame to its optimal size
        pack();
    }
}
```

**Figure 9.44** `FrameWithHelloCanvas.java`

As mentioned, it is necessary to set the size of a `Canvas` object explicitly by calling the `setSize()` method of the `Canvas` object, or it is invisible by default.

A driver program, `TestFrameWithHelloCanvas` class, can be found in the course CD-ROM and website. Compile the classes, `HelloCanvas`, `FrameWithHelloCanvas` and `TestFrameWithHelloCanvas`. Execute the `TestFrameWithHelloCanvas`, and a GUI like the one in Figure 9.45 appears on the screen.

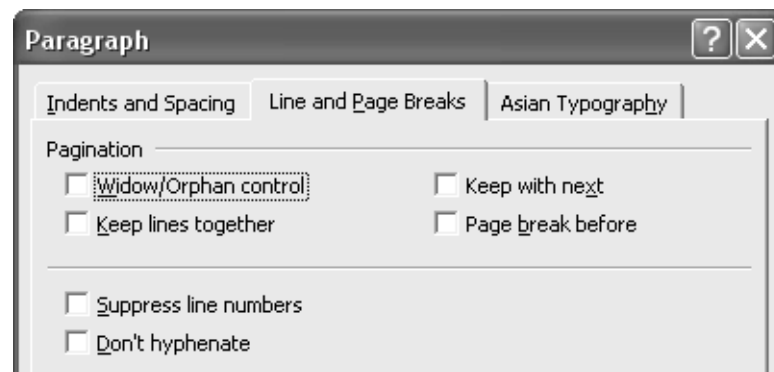


**Figure 9.45** A GUI with a canvas object modelled by the `HelloCanvas` class

The appearance of the `HelloCanvas` is fixed, but a canvas is usually expected to construct its appearance according to some data that can be changed during software execution. Appendix H of the unit discusses how to write canvas-like drawing programs. The techniques discussed in the appendix can also be applied to write a subclass of the `Canvas` class. As the program involves the techniques of event handling, discussed later in this unit, you can review that appendix after you have completed the section on event handling.

## Checkboxes

A checkbox on a GUI is a two-state widget that is either checked (selected) or unchecked (cleared) and is therefore frequently used for obtaining two-state entries, such as yes/no, true/false, disable/enable and so on. For example, in Microsoft Word, the Line and Page Breaks tab of the Paragraph dialog uses several checkboxes to get the options from the users, as shown in Figure 9.46.



**Figure 9.46** The Line and Page Breaks tab of the Paragraph dialog in Microsoft Word uses checkboxes to obtain a user's selection

The Swing package provides the `JCheckBox` class that models checkboxes. To create a `JCheckBox` object, the following three constructors are frequently used:

```
public JCheckBox()  
public JCheckBox(String text)  
public JCheckBox(String text, boolean selected)
```

The first constructor creates a checkbox without text. For the latter two constructors, the first supplementary data supplied to the constructor is a `String` object whose content is used as the text of the checkbox. Initially, the checkbox is by default unchecked. To create a `JCheckBox` object that is initially checked, use the third constructor and provide a boolean value of `true` as the second supplementary data.

The `JCheckBox` class inherits the following two methods from its superclass `javax.swing.AbstractButton` (or simply `AbstractButton`)

```
public boolean isSelected(), and  
public void setSelected(boolean b)
```

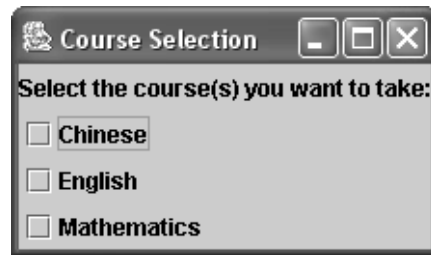
for getting and setting the state of a `JCheckBox` object programmatically.

A class `CourseSelectionFrame` that illustrates how to use `JCheckBox` objects as GUI components is shown in Figure 9.47.

```
// Resolve classes in java.awt and javax.swing packages  
import java.awt.*;  
import javax.swing.*;  
  
// Class definition of class CourseSelectionFrame  
public class CourseSelectionFrame extends JFrame {  
    // The checkboxes to be shown  
    private JCheckBox chinese = new JCheckBox("Chinese");  
    private JCheckBox english = new JCheckBox("English");  
    private JCheckBox math = new JCheckBox("Mathematics");  
  
    // Constructor  
    public CourseSelectionFrame() {  
        // Set the frame title  
        super("Course Selection");  
  
        // Get the content pane from the JFrame object  
        Container contentPane = getContentPane();  
  
        // Set the layout of the content pane to be GridLayout with  
        // 3 rows and 1 column  
        contentPane.setLayout(new GridLayout(3, 1));  
  
        // Add the checkboxes to the content pane  
        contentPane.add(chinese);  
        contentPane.add(english);  
        contentPane.add(math);  
  
        // Set the default operation when the close button of the  
        // frame is clicked  
        setDefaultCloseOperation(EXIT_ON_CLOSE);  
  
        // Set the GUI to the optimal size  
        pack();  
    }  
}
```

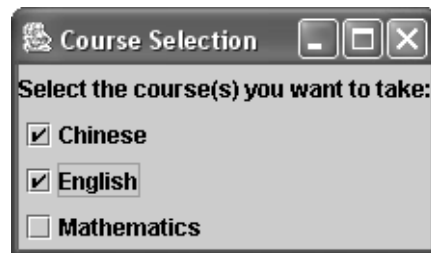
**Figure 9.47** `CourseSelectionFrame.java`

A driver program, `TestCourseSelectionFrame`, is written and is available on the course CD-ROM and website. Compile the classes and execute the `TestCourseSelectionFrame`. A GUI like the one in Figure 9.48 is shown on the screen.



**Figure 9.48** A GUI that uses `JCheckBox` for getting user entries

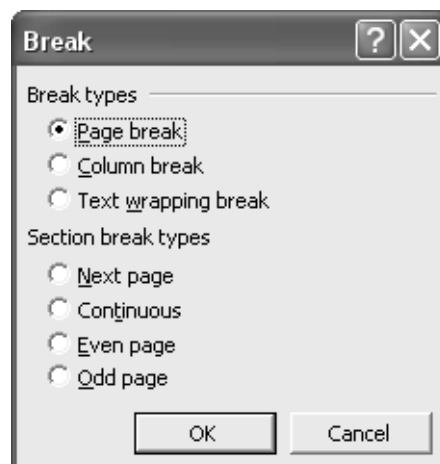
Clicking the checkbox or the associated text will toggle the state of the checkbox. For example, the Chinese and English checkboxes in the `CourseSelectionFrame` are checked and shown in Figure 9.49.



**Figure 9.49** The `CourseSelectionFrame` GUI with Chinese and English checkboxes checked

## Radio boxes

Similar to a checkbox, a radio box is a two-state GUI component but is usually used as a group of radio boxes. At any time, only one radio box in the group is selected. That is, selecting a radio box in the group unselects the currently selected one. For example, the Break dialog of the Microsoft Word has a group of radio boxes to be selected by the user for the desired break type.



**Figure 9.50** The Break dialog of Microsoft Word uses radio boxes to obtain the user's selection

Radio boxes are modelled by the class `javax.swing.JRadioButton` (or simply `JRadioButton`). The ways to create them are similar to those of `JCheckBox`. The following three are commonly used overloaded constructors:

```
public JRadioButton()  
public JRadioButton(String text)  
public JRadioButton(String text, boolean selected)
```

The first constructor creates a `JRadioButton` object without text — just the radio box itself. For the latter two constructors, the first supplementary data supplied to the constructors is the reference to a `String` object whose content is used as the radio box text. To set a radio button to be selected initially, use the third constructor to create the `JRadioButton` object and provide a boolean value of `true` as the second supplementary data.

After the `JRadioButton` objects have been created, they do not know they are included in the same radio box group. To relate and manage a group of `JRadioButton` objects, it is necessary to create a helper object, a `ButtonGroup` object. For all `JRadioButton` objects that logically belong to the same group, call the `add()` method of the `ButtonGroup` object with supplementary data to reference a `JRadioButton` object one at a time.

`JRadioButton` is another subclass of the `AbstractButton` class. It therefore inherits the two methods

```
public boolean isSelected(), and  
public void setSelected(boolean b)
```

for getting and setting the state of a `JRadioButton` object.

The `RegionSelectionFrame` as shown in Figure 9.51 enables the user to select a region using radio boxes.



**Figure 9.51** Radio boxes are used for getting a user's selection of region

The definition of the class is provided in Figure 9.52.

```
// Resolve classes in java.awt and javax.swing packages
import java.awt.*;
import javax.swing.*;

// Definition of class RegionSelectionFrame
public class RegionSelectionFrame extends JFrame {
    // GUI components
    private JLabel label = new JLabel("Select the region:");
    private ButtonGroup optionGroup = new ButtonGroup();
    private JRadioButton hkButton =
        new JRadioButton("Hong Kong Island", true);
    private JRadioButton klnButton = new JRadioButton("Kowloon");
    private JRadioButton ntButton = new JRadioButton("New Territories");

    // Constructor
    public RegionSelectionFrame() {
        // Set the frame title
        super("Region Selection");

        // Get the content pane of the frame and set a FlowLayout object
        // to be its layout manager
        Container contentPane = getContentPane();
        contentPane.setLayout(new GridLayout(4, 1));

        // Add the GUI components to the frame
        contentPane.add(label);
        contentPane.add(hkButton);
        contentPane.add(klnButton);
        contentPane.add(ntButton);

        // Associate all radio buttons by a ButtonGroup object
        optionGroup.add(hkButton);
        optionGroup.add(klnButton);
        optionGroup.add(ntButton);

        // Set the default operation when the close button of the frame
        // is clicked
        setDefaultCloseOperation(EXIT_ON_CLOSE);

        // Set the GUI to its optimal size
        pack();
    }
}
```

**Figure 9.52** RegionSelectionFrame.java

Of the three `JRadioButton` objects created by a `RegionSelectionFrame` object, the `JRadioButton` object for the “Hong Kong Island” option is created using the constructor with two supplementary data of a `String` object and a boolean value of `true`. The other two `JRadioButton` objects are created by supplying a



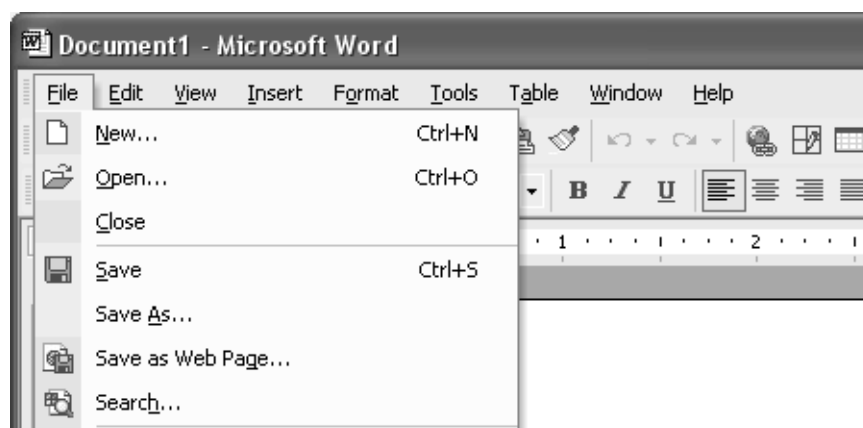
String object only. Such an approach sets the “Hong Kong Island” option as the initially selected one.

After the `JRadioButton` objects are added to the content pane of the frame object, it is necessary to denote that they are in the same radio box group by adding them to the `ButtonGroup` object referred to by the variable `optionGroup`.

A driver program, `TestRegionSelectionFrame`, is written and is available on the course CD-ROM and website. Compile the classes and execute the `TestRegionSelectionFrame` program. The GUI shown in Figure 9.51 above will appear on the screen.

## Menu bars, menus and menu items

For some complicated software applications, it is often impossible to provide buttons for all functionalities on the GUIs. Then, it is common for functionalities to be categorized and provided on a menu bar. For example, most functionalities of Microsoft Word are accessible from the menu bar as shown in Figure 9.53.



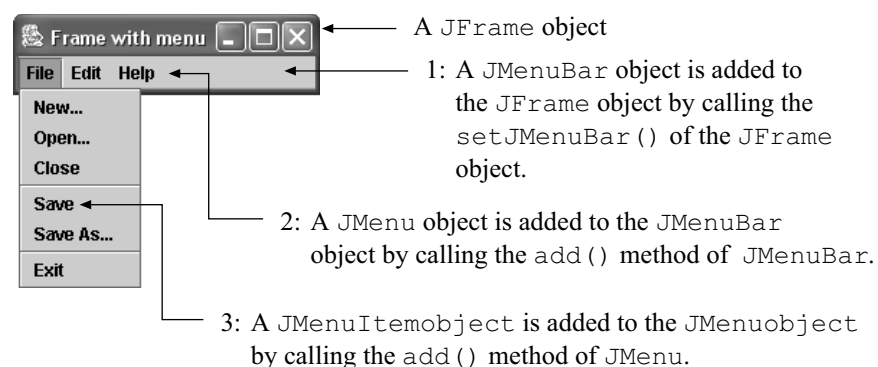
**Figure 9.53** A menu bar of Microsoft Word

A GUI built with the Java standard software library can also be equipped with a menu bar so that the functionalities provided by the GUI are systematically arranged. The steps in adding a menu bar to a `JFrame` are:

- 1 Create a `JMenuBar` object and add it to a `JFrame` object by calling the `setJMenuBar()` method of the `JFrame` with the reference of the `JMenuBar` object.
- 2 For each menu on the menu bar, create `JMenu` objects (using the constructor `JMenu()` with supplementary data of a `String` object of the menu text). Add them to the `JMenuBar` object one by one by calling the `add()` method of the `JMenuBar` with the reference of a `JMenu` object. The order of adding the `JMenu` objects to the `JMenuBar` determines the order of menus (starting from the left) on the menu bar.

- 3 Clicking the menu on the menu bar will usually show a drop-down menu list. Clicking a menu item in the list will trigger the software application to perform some predefined operations. To add menu items to a menu, create `JMenuItem` objects (using the constructor `JMenuItem()` with supplementary data of a `String` object of the menu item text). Add them to the `JMenu` object one by one by calling the `add()` method of the `JMenu` with the reference of a `JMenuItem` object. The order of adding the `JMenuItem` objects to the `JMenu` object determines the order of menu items (starting from the top) on the menu. If you want to add a line separating the menu items on a menu, you can call the `addSeparator()` method of the `JMenu` object in between adding other `JMenuItem` objects.

The above steps are visualized in Figure 9.54.



**Figure 9.54** The relationships among `JFrame`, `JMenuBar`, `JMenu` and `JMenuItem`

Adding a menu to a `JFrame` object does not affect the content pane. It is therefore possible to add other GUI components to the `JFrame` content pane as usual. To construct the menu system of the GUI as shown in Figure 9.54, a class `FrameWithMenu` is written in Figure 9.55.

```
// Resolve classes in javax.swing package
import javax.swing.*;

// Definition of class FrameWithMenu
public class FrameWithMenu extends JFrame {
    // Prepare the object for the menu bar
    private JMenuBar menuBar = new JMenuBar();
    // Prepare the object for the menu
    private JMenu fileMenu = new JMenu("File");
    private JMenu editMenu = new JMenu("Edit");
    private JMenu helpMenu = new JMenu("Help");
    // Prepare the menu items for file menu
    private JMenuItem newMenuItem = new JMenuItem("New...");
    private JMenuItem openMenuItem = new JMenuItem("Open...");
    private JMenuItem closeMenuItem = new JMenuItem("Close");
    private JMenuItem saveMenuItem = new JMenuItem("Save");
    private JMenuItem saveAsMenuItem = new JMenuItem("Save As...");
    private JMenuItem exitMenuItem = new JMenuItem("Exit");
    // Prepare the menu items for the edit menu
    private JMenuItem cutMenuItem = new JMenuItem("Cut");
    private JMenuItem copyMenuItem = new JMenuItem("Copy");
    private JMenuItem pasteMenuItem = new JMenuItem("Paste");
```

```

        // Prepare the menu items for the help menu
        private JMenuItem helpMenuItem =
            new JMenuItem("FrameWithMenu Help");
        private JMenuItem aboutMenuItem =
            new JMenuItem("About FrameWithMenu");

        // Constructor
        public FrameWithMenu() {
            // Set the frame title
            super("Frame with menu");

            // Set the menu bar
            setJMenuBar(menuBar);

            // Add the menu to the menu bar
            menuBar.add(fileMenu);
            menuBar.add(editMenu);
            menuBar.add(helpMenu);

            // Add the menu items to the file menu
            fileMenu.add(newMenuItem);
            fileMenu.add(openMenuItem);
            fileMenu.add(closeMenuItem);
            fileMenu.addSeparator();
            fileMenu.add(saveMenuItem);
            fileMenu.add(saveAsMenuItem);
            fileMenu.addSeparator();
            fileMenu.add(exitMenuItem);

            // Add the menu items to the edit menu
            editMenu.add(cutMenuItem);
            editMenu.add(copyMenuItem);
            editMenu.add(pasteMenuItem);

            // Add the menu items to the help menu
            helpMenu.add(helpMenuItem);
            helpMenu.addSeparator();
            helpMenu.add(aboutMenuItem);

            // Set the default operation of the frame when the close button
            // is clicked
            setDefaultCloseOperation(EXIT_ON_CLOSE);

            // Set the frame to its optimal size
            pack();
        }
    }

```

**Figure 9.55** FrameWithMenu.java

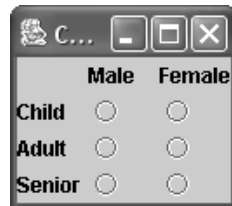
A driver program, `TestFrameWithMenu`, is written and is available on the course CD-ROM and website. Compile the classes and execute the `TestFrameWithMenu`. The GUI in Figure 9.54 above will be shown on the screen.

You have now learned how to create commonly used GUI components. Before proceeding to the next section discussing building complex GUIs,

please use the following self-test to test your understanding of creating GUIs with the different GUI components we discussed in this section.

### Self-test 9.3

Please write classes that build the following GUIs and the corresponding driver programs that show the GUIs on the screen.



	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9
2	2	4	6	8	10	12	14	16	18
3	3	6	9	12	15	18	21	24	27
4	4	8	12	16	20	24	28	32	36
5	5	10	15	20	25	30	35	40	45
6	6	12	18	24	30	36	42	48	54
7	7	14	21	28	35	42	49	56	63
8	8	16	24	32	40	48	56	64	72
9	9	18	27	36	45	54	63	72	81

Hint: To set the alignment of the text on a `JLabel` object to be center, create the `JLabel` object using an overloaded constructor:

```
public JLabel(String text, int horizontalAlignment)
```

Please refer to the API documentation of the `JLabel` class.

## Dialogs

A dialog is a special type of frame shown on the screen either for getting user input or for conveying a particular message that needs the user's immediate attention. As dialogs are frequently used GUI widgets, especially standard input and message dialogs, the Swing packages provide the `javax.swing.JOptionPane` class (or simply the `JOptionPane` class) for creating and using the standard dialogs. You have been using them since *Unit 4*. See Appendix D for more information.

## Creating complex GUIs

Using the three layout manager classes we discussed, you can create some basic GUI applications. However, the complexities of the GUI are limited by the capability of individual layout manager classes. Practically, most GUIs, such as the calculator applications shown in Figure 9.1 and Figure 9.2, are far more complex and cannot be built using a layout manager class.

To create complex GUIs, you have two options.

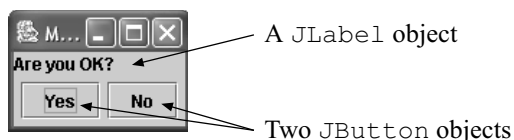
- 1 The first one is to build a GUI without a layout manager object. The GUI components are added to the GUI with sizes and locations that you specify. Then, you are free to arrange the GUI components on the GUI. There are several problems with this approach. The first is that the GUIs become platform dependent, because you can probably only set the sizes and locations of the GUI components for a particular window system. If the GUI is to be shown on another window system, the sizes and locations of the components are no longer optimal. Another even more significant problem is that the components are not automatically rearranged if the GUI is resized.
- 2 Another approach, which is preferable, is to build the GUIs with panels. This is discussed in the following subsection.

### Another container: `JPanel`

A panel (modelled by the `javax.swing.JPanel` class) is also a GUI component that can be added to a container object. Furthermore, a panel itself is also a container in which other GUI components can be added. By making use of the panels, you can develop complex GUIs.

### Complex GUIs with panels

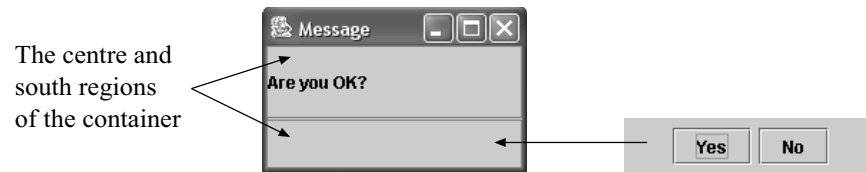
For example, a `MessageFrame` GUI shown in Figure 9.56 cannot be built by simply using a container with a layout manager.



**Figure 9.56** A user-defined message frame

The `MessageFrame` is built using two containers with a `BorderLayout` object and `FlowLayout` object as their layout managers respectively. Then, the content pane of the frame can be set to use a `BorderLayout` object as its layout manager, so that the `JLabel` object can be added to its center region. The two `JButton` objects can be added to another container that is a `JPanel` object, and such a `JPanel` object can be treated

as a usual GUI component to be added to the south region of the content pane of the frame. Such an arrangement is visualized in Figure 9.57.



**Figure 9.57** The architecture of the `MessageFrame` with a `JPanel` object in the south region

The default layout manager for a `JPanel` is a `FlowLayout` object. The `JPanel` object to be added to the south region of the `MessageFrame` is therefore simply:

```
JPanel buttonPanel = new JPanel();
buttonPanel.add(okButton);
buttonPanel.add(cancelButton);
```

Then, the `JPanel` object can be added to the content pane of the `MessageFrame` object as usual.

```
contentPane.add(buttonPanel, BorderLayout.SOUTH);
```

The complete definition of the class `MessageFrame` is shown in Figure 9.58.

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

// Definition of class MessageFrame
public class MessageFrame extends JFrame {
    // Attributes
    private JLabel messageLabel = new JLabel();
    private JButton okButton = new JButton("Yes");
    private JButton cancelButton = new JButton("No");

    // Constructor
    public MessageFrame(String message) {
        // Provide a String to the superclass constructor
        // to set the frame title
        super("Message");

        // Get the content pane of the frame for adding GUI components
        Container contentPane = getContentPane();

        // Create a button panel
        JPanel buttonPanel = new JPanel();
        buttonPanel.add(okButton);
        buttonPanel.add(cancelButton);

        // Add the label and the button panel to the content pane
        contentPane.add(messageLabel, BorderLayout.CENTER);
        contentPane.add(buttonPanel, BorderLayout.SOUTH);
    }
}
```

```

        // Set the label text according to the constructor parameter
        messageLabel.setText(message);

        // Set the default behaviour of clicking the close button
        setDefaultCloseOperation(EXIT_ON_CLOSE);

        // Set the frame to its optimal size
        pack();
    }
}

```

**Figure 9.58** MessageFrame.java

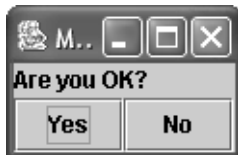
To test the `MessageFrame` class, a `TestMessageFrame` class is written. Please refer to the course CD-ROM and website for the definition.

Similar to the content pane of a frame, you can set different layout managers to it, and the GUI components added to a `JPanel` can be other `JPanel` objects. As a result, even more complex GUIs can be built with `JPanel` objects.

Please use the following self-test to test your ability to build complex GUIs.

### Self-test 9.4

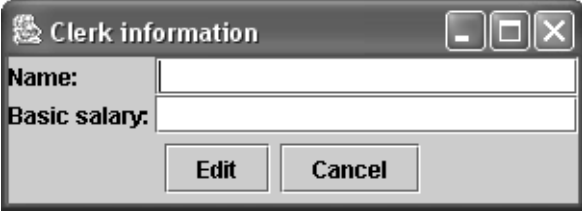
- 1 Modify the `MessageFrame` class so that the `JPanel` object uses a `GridLayout` object as its layout manager. Then, the `MessageFrame` becomes:



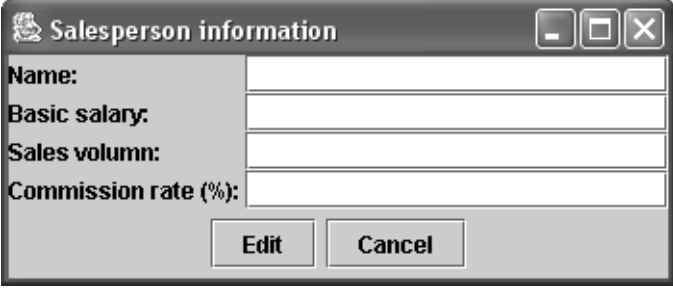
- 2 Write the class `TicTacToeFrame` with the following appearance and write a driver program `TestTicTacToeFrame` to test it.



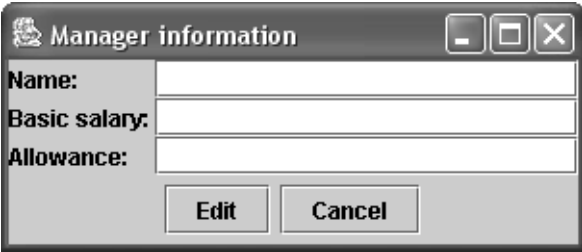
3 Write classes for the following frames:



A Java Swing dialog box titled "Clerk information" with a standard icon. It contains two text input fields: "Name:" and "Basic salary:". Below the fields are two buttons: "Edit" and "Cancel".



A Java Swing dialog box titled "Salesperson information" with a standard icon. It contains four text input fields: "Name:", "Basic salary:", "Sales volumn:", and "Commission rate (%):". Below the fields are two buttons: "Edit" and "Cancel".



A Java Swing dialog box titled "Manager information" with a standard icon. It contains three text input fields: "Name:", "Basic salary:", and "Allowance:". Below the fields are two buttons: "Edit" and "Cancel".

You can create individual classes as the subclasses of the `JFrame` class for the above frames. Furthermore, you can investigate the similarities among the three GUIs and find a way to reduce duplicated codes by inheritance.

---



## Event-handling model

We discussed how to build GUIs with different GUI components and different layout managers. When those GUIs are shown on the screen, you can click the buttons and enter text in the textboxes. However, clicking a button performs no operations. Furthermore, when the <Close> button of a frame is clicked, it is sometimes necessary to prompt the user to save the changes made to the data being processed.

The actions taken by a user are considered by the GUI to be *events*, such as clicking a button, moving the mouse pointer within the GUI, and clicking the <Minimize> or <Maximize> button of a frame. All these actions are encapsulated as event objects.

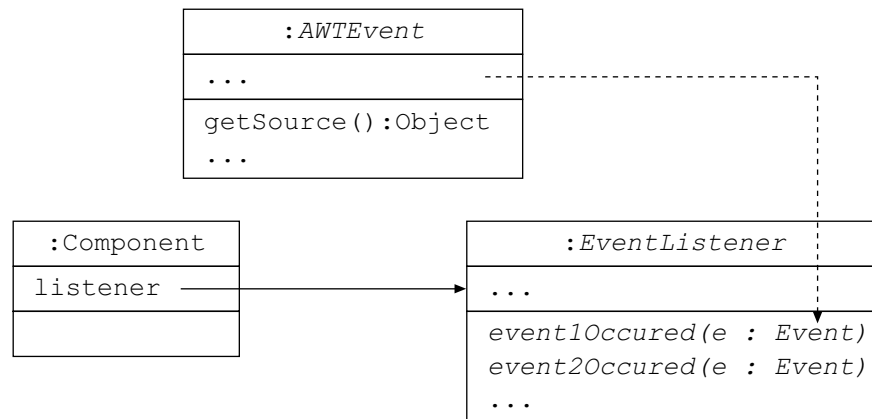
We discuss the life cycle of an event and some common events, in the following subsections.

### Event sources, events and event listeners

Similar to exception handling that we discussed in *Unit 8*, event objects are created when a user interacts with a GUI. For all GUIs we have discussed so far in this unit, event objects are created, but the programs are not written to handle those events or event objects. The program ignores all events.

For whichever GUI component that handles an event, it is necessary to register an object of a suitable type, which is known as an event handler or an event listener that handles the event, with the GUI components. That is, call a method of the GUI component with the reference of the event-handler object as supplementary data, so that the GUI component has the reference of the registered event handler. When a user operates the GUI, such as clicking a button using the mouse pointer, an event object that encapsulates all information about the event is created, and a particular method of the registered event handler is called with the reference of the created event object. Then, the method can determine the information about the event that occurred and perform operations accordingly.

Three objects are involved the entire process: the button object, the created event object and the registered event handler object. The button object that causes the event is called as the event source. The scenario is visualized in Figure 9.59.



**Figure 9.59** The relationship among event source object, event object and event handler object

In Figure 9.59, all class names are set in italics because they are abstract. The *EventListener* object has been registered with the *Component* object, so that the *Component* object maintains the reference of the *EventListener* object, as indicated by the solid arrow. When an event occurs with the component as the event source, an event object is created and all registered event listener objects are called with the event object as the supplementary data, which is denoted by the dashed arrow.

The events that can be caused are classified, and listener classes are defined for each category, so that each method defined in a listener class corresponds to an event type in the category. Table 9.2 shows all the event listener classes and their methods.

Eleven event listener classes are listed in Table 9.2, corresponding to 11 common event categories. Furthermore, you can see that the naming convention used for the method names is different from the usual functions. We said that method names are usually verbs, as they perform a particular operation. The methods defined in the above listener classes are mostly named according to the event that occurred, such as the *actionPerformed()* method. Each method accepts supplementary data as the reference to an event object that encapsulates the information about the event that occurred. Later in the unit, we discuss how to handle some of the above-mentioned event categories.

Listener classes presented in Table 9.2 are not usual classes but Java interfaces. Before we proceed to the discussions on handling common events, we have to discuss what Java interfaces are.

**Table 9.2** Common GUI listener classes and the defined methods

Category	Listener type name	Method defined
Action	ActionListener	void actionPerformed(ActionEvent e)
Adjustment	AdjustmentListener	void adjustmentValueChanged(AdjustmentEvent e)
Component	ComponentListener	void componentHidden(ComponentEvent e) void componentMoved(ComponentEvent e) void componentResized(ComponentEvent e) void componentShown(ComponentEvent e)
Container	ContainerListener	void componentAdded(ContainerEvent e) void componentRemoved(ContainerEvent e)
Focus	FocusListener	void focusGained(FocusEvent e) void focusLost(FocusEvent e)
Item	ItemListener	void itemStateChanged(ItemEvent e)
Key	KeyListener	void keyPressed(KeyEvent e) void keyReleased(KeyEvent e) void keyTyped(KeyEvent e)
Mouse	MouseListener	void mouseClicked(MouseEvent e) void mouseEntered(MouseEvent e) void mouseExited(MouseEvent e) void mousePressed(MouseEvent e) void mouseReleased(MouseEvent e)
Mouse motion	MouseMotionListener	void mouseDragged(MouseEvent e) void mouseMoved(MouseEvent e)
Text	TextListener	void textValueChanged(TextEvent e)
Window	WindowListener	void windowActivated(WindowEvent e) void windowClosed(WindowEvent e) void windowClosing(WindowEvent e) void windowDeactivated(WindowEvent e) void windowDeiconified(WindowEvent e) void windowIconified(WindowEvent e) void windowOpened(WindowEvent e)

## Interface

### What are Java interfaces?

In *Unit 7*, we said that Java abstract classes have abstract methods without method bodies that are to be provided or implemented by their subclasses. What if all the methods defined in an abstract class are abstract? The implication is that the class defines no method implementations and all method bodies are to be implemented by its

subclasses. You can imagine that such an abstract class with all methods abstract can be used to define the methods its subclasses have to implement, and a variable of such type can be used to refer to objects of any subclasses of the abstract superclass.

If all methods of a class are abstract, it is preferable to define them as an interface in the Java programming language. The definition of a Java interface is:

```
[public] interface interface-name {  
    [ class-variable-declaration(s) ]  
    [ abstract-method-declaration(s) ]  
}
```

The access modifier `public` is optional. Like a class definition, if an interface is marked `public`, the classes of other packages can access it. Otherwise, only classes of the same package can access it.

All rules governing the naming conventions of classes are applicable to interfaces. As an interface also defines a type, its name is usually a noun. Both variable and method declarations are optional. All variable declarations of an interface are by default `public`, `static` and `final`. In other words, you can define universally accessible constant class variables in an interface only. Furthermore, all method declarations are understood to be `public` and `abstract`. It is optional but good practice to provide these two keywords to a method definition of an interface.

Java interfaces are usually used for defining a type and the messages it can accept, that is, the method names with parameter names and return types. In other words, a Java interface defines the methods or behaviours its subclass must possess because all methods defined by a Java interface are `abstract`, and subclasses of an interface must define all methods declared by the interface, or they are kept `abstract`. Further discussion on Java interfaces is provided in Appendix E. Please read it to have a better understanding before proceeding to the next subsection for event handling.

## Implementing event handlings

Table 9.2 presents some common event types, but they are not applicable to all GUI components. For example, only `Container` class and its subclasses support the `Container` event category. To find out the event categories supported by a particular GUI component, you can consult the API documentation.

If a class supports an event category, you can find that the class or its superclass defines the method

```
public void addCategoryListener(CategoryListener l)
```

for registering the supplied object, whose class implements the interface `CategoryListener`, to be the event listener object that will handle all

events in the category if an event specified by the category occurs. As a usual practice, a class that defines an `addXXX()` method usually defines a `removeXXX()` method for removing the supplied object from it. Therefore, the class usually defines another method:

```
public void removeCategoryListener(CategoryListener l)
```

For event handling, the pair of add/remove methods hints that a GUI component can register more than one listener. And when an event of the category occurs, the corresponding handler methods of all registered event handler objects are executed one by one.

You can see that the add/remove methods expect supplementary data of the reference of an object whose class implements the *CategoryListener* interface. Therefore, the class of the supplied object has to implement all methods specified by the interface, or an object the class cannot be created.

The reason Java interface is used as the type of object to be supplied to the add/remove methods is that it mandates the handler class to implement all methods declared in the listener interface.

For example, most GUI components support the Action category event. You can find that most GUI component classes define the pair of add/remove methods for *ActionListener* objects:

```
public void addActionListener(ActionListener l)
public void removeActionListener(ActionListener l)
```

The steps in setting up event handling for a GUI are:

- 1 Identify the category of the event the GUI component is expected to handle.
- 2 Write an event handler class that implements the listener interface.
- 3 While constructing the GUI, create the GUI component and an object of the event handler class.
- 4 Call the `addXXX()` method of the GUI component to register the event handler object to be its event handler.

Then, if the GUI component causes an event, an event object is created and the corresponding method of the event handler object is called with the reference to the event object as the supplementary data. For example, for the class implementing the *ActionListener* interface, the `actionPerformed()` method will be called with the reference to an *ActionEvent* object as supplementary data.

In the following subsection, we discuss the most common event category — Action — by using the *ActionListener* interface and *ActionEvent* event. Two other common event categories are provided in Appendix F.

## Action events

Most GUI components support the Action category event. The interpretation of such a category varies with respect to different GUI components. When the handler method is called, the reference to an `ActionEvent` object is passed to the method as well. As mentioned, the event object encapsulates the information about the event that occurred. For an `ActionEvent` object, you will find two pieces of information useful. The first one is the event source that caused the event. The other is an action command obtained by calling the two methods, `getSource()` and `getActionCommand()` respectively. Table 9.3 provides the operations that correspond to Action events for different GUI components and the contents of the action command obtained by calling the `getActionCommand()` method of the `ActionEvent` object.

**Table 9.3** Interpretation of Action event for common GUI components

GUI component	Operation considered to be Action event	Default contents of the action command
<code>JButton</code>	The button is clicked.	The text on the <code>JButton</code> object.
<code>JTextField</code>	The <Enter/Return> key is pressed.	The text entered into the <code>JTextField</code> object by the user.
<code>JMenuItem</code>	The menu item is selected.	The title of the <code>JMenuItem</code> .
<code>JCheckbox</code>	The checkbox is clicked.	The title of the checkbox.
<code>JRadioBox</code>	The radio box is clicked.	The title of the radio box.

From Table 9.3, you can see that handling the Action category event is sufficient for typical use of the GUI components. Now, let's have a look at a use of handling Action events caused by a `JButton` object.

To handle an Action category event, you first have to write a class that is a subclass of the interface `ActionListener`. The following is the definition of the interface `ActionListener`.

<i><code>ActionListener</code></i>
<i><code>+actionPerformed(ActionEvent ae)</code></i>

(The class name and the method name are set in italics because they are abstract.)

Based on the interface definition, the definition of the handler class should look like:

```
[public] class class-name [extends superclass-name]
    implements ActionListener {
    ...
    public void actionPerformed(ActionEvent ae) {
        ...
    }
    ...
}
```

## Event handling by a dedicated object

With what we have discussed, we can write an event handler class for handling a frame with a button on it, as shown in Figure 9.60.



**Figure 9.60** A frame with a button that can respond to a mouse click

First of all, the handler class, `ActionHandler1`, is written in Figure 9.61.

```
// Resolve interface ActionListener in the java.awt.event package
import java.awt.event.*;

// Definition of class ActionHandler1
public class ActionHandler1 implements ActionListener {

    // The event handler method
    public void actionPerformed(ActionEvent ae) {
        System.out.println(
            "The button [" + ae.getActionCommand() + "] is clicked.");
    }
}
```

**Figure 9.61** `ActionHandler1.java`

Then, the class `GUIWithEvent1` that creates the frame shown in Figure 9.60 above is shown in Figure 9.62.

```
// Resolve classes used in the class definition
import java.awt.BorderLayout;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;

// Definition of class GUIWithEvent1
public class GUIWithEvent1 extends JFrame {
    // Attribute
    private JButton button = new JButton("Click Me");
```

```

public GUIWithEvent1() {
    // Set the frame title
    super("GUI with Event");

    // Get the content pane and add a button to it
    getContentPane().add(button, BorderLayout.CENTER);

    // Create an ActionListener object as the event handler
    // object
    ActionListener listener = new ActionListener1();

    // Register the event handler object to be the event handler
    // of the button
    button.addActionListener(listener);

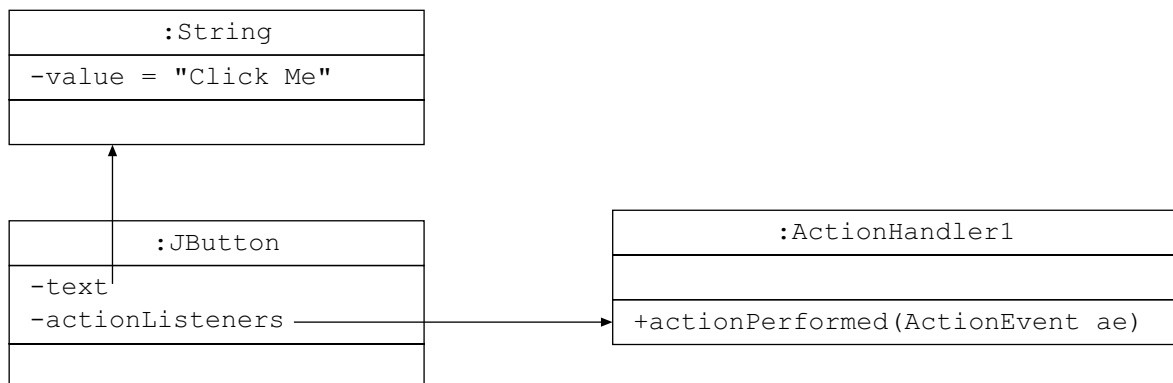
    // Set the default operation when the close button is
    // clicked
    setDefaultCloseOperation(EXIT_ON_CLOSE);

    // Set the GUI to the optimal size
    pack();
}
}

```

**Figure 9.62** GUIWithEvent1.java

The definition of the class `GUIWithEvent1` is similar to the definition of `FrameWithButton2` (Figure 9.17), except for the program segment for creating an event handler object and registering it with the `JButton` object. After the constructor of a `GUIWithEvent1` is executed, the scenario of the created object is visualized in Figure 9.63. (The definitions of the classes in Figure 9.63 may not be implemented as shown. They are presented for clarification only.)

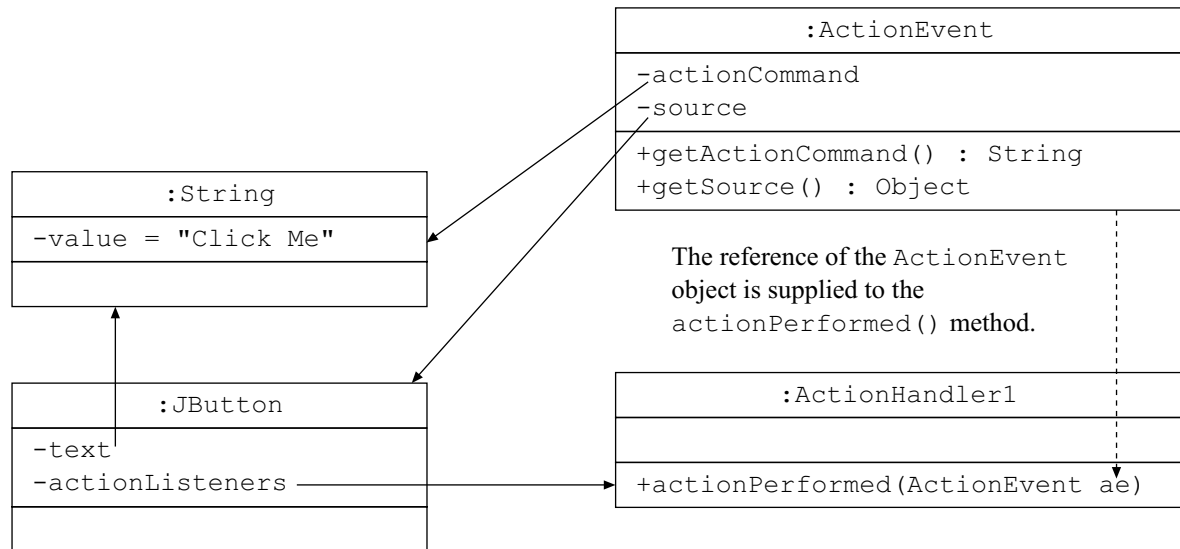
**Figure 9.63** The scenario of the objects after registering the `ActionHandler1` object as the handler object of the `JButton` object

The `JButton` object has a reference to the string object holding the text on it and another reference to the action handler object.

Whenever the `JButton` object is clicked, an `ActionEvent` object is created that specifies the `JButton` object as the event source using the



variable `source` and the text on it as the action command using the variable `actionCommand`, as shown in Figure 9.64.



**Figure 9.64** The scenario when the `JButton` is clicked

The reference of the created `ActionEvent` object is supplied to `ActionHandler1` as the supplementary data (as parameter `ae`) of the method call `actionPerformed()`. Notice a dashed arrow is used in Figure 9.64 to indicate this is an action. The references to objects are the same as those in Figure 9.63. Then, in the `actionPerformed()` method, it is possible to obtain this information for further manipulation. For the `ActionHandler1` object, the following `actionPerformed()` method is executed:

```
public void actionPerformed(ActionEvent ae) {
    System.out.println(
        "The button [" + ae.getActionCommand() + "] is clicked.");
}
```

Therefore, the following output is shown on the screen every time the `JButton` object is clicked:

```
The button [Click Me] is clicked.
```

A driver program `TestGUIWithEvent1` is written to test the `GUIWithEvent1` class. Please find it on the course CD-ROM and website.

With the `getSource()` method, you can get the reference of the GUI component that caused the event. You should notice that the return type of the `getSource()` method is `java.lang.Object` and the returned reference can refer to any GUI component. Therefore, if you want to manipulate the event source object, you should cast the object reference properly before accessing the referred object. For example, another event handler class `ActionHandler2` that implements the `ActionListener` is written in Figure 9.65.

```

// Resolve classes in java.awt.event for event handling
import java.awt.event.*;
// Resolve the JButton class in javax.swing package
import javax.swing.*;

// Definition of class ActionHandler2
public class ActionHandler2 implements ActionListener {
    private int clickCount = 0;

    public void actionPerformed(ActionEvent ae) {
        Object source = ae.getSource();
        if (source instanceof JButton) {
            JButton button = (JButton) source;
            clickCount++;
            button.setText("Clicked : " + clickCount);
        }
    }
}

```

**Figure 9.65** ActionHandler2.java

Another class GUIWithEvent2 is written that uses an ActionHandler2 object as its event handler for an Action category event. The definition is exactly the same as the GUIWithEvent1 except for the following statement:

```

ActionListener listener = new ActionHandler2();

```

The driver program, TestGUIWithEvent2 is also similar to that of TestGUIWithEvent1 except the object to be manipulated is a GUIWithEvent2 object instead of a GUIWithEvent1 object. For the complete definitions of these two classes, please refer to the course CD-ROM or website.

Compile the classes and execute the TestGUIWithEvent2 class. The GUI is similar to that of the GUI shown by TestGUIWithEvent1 (Figure 9.60). When the button on the frame is clicked, an ActionEvent object is created and is supplied to the registered listener object for the Action category event, the ActionHandler2 object, while its actionPerformed() method is called.

First of all, the reference of the source object is obtained by calling the getSource() method of the ActionEvent object.

```

Object source = ae.getSource();

```

Although it is quite sure that the object that caused the Action event is a JButton for the GUIWithEvent2 object, it is always good programming practice to verify whether the event source is a JButton by an if statement with the instanceof operator used in its condition. If the object is verified to be a JButton object, the reference is safely casted to JButton type and is assigned to the variable button.

```

if (source instanceof JButton) {
    JButton button = (JButton) source;
    ...
}

```

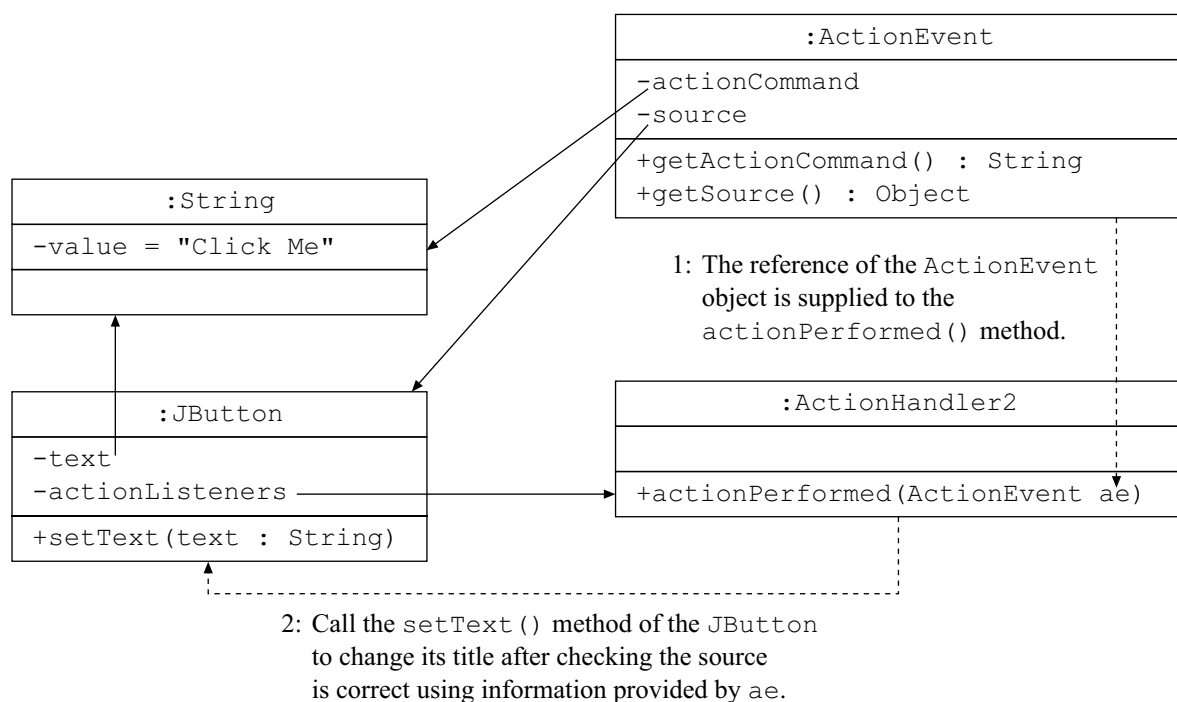
Then, it is then possible to increase the counter for the button being clicked, the variable `clickCount`, and a title is prepared and set to be the `JButton` text.

```

clickCount++;
button.setText("Clicked : " + clickCount);

```

The above-mentioned sequence of operations can be summarized by step 2 in Figure 9.66.



**Figure 9.66** The scenario when the `JButton` is clicked and the sequence of messages sent

Notice that step 2 is an action and is indicated by a dashed arrow. The significance of such an approach is that a dedicated object is created with methods that are solely for handling the events. Instead of creating a dedicated object to handle the event, it is possible to equip the GUI container object with the ability to handle the events, as discussed in the following subsection.

## Event handling by the GUI container object

Although using a dedicated object that is solely for event handling is simpler, there is a limitation to the event-handling approach using a dedicated event handler object. The event-handler method can only get the reference of the GUI component that caused the event. That is, it is not possible for the event handler method to get the references of the

GUI container and other GUI components; hence it limits the functionality of the event handler method.

Instead, it is more appropriate to assign the GUI container object, which has the references of other GUI components, the role to handle events. For example, as a variation of the GUIWithEvent2 class, a class GUIWithEvent3 is written to be the GUI container and the event handler as shown in Figure 9.67.

```
// Resolve classes used in the class definition
import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;

// Definition of class GUIWithEvent3
public class GUIWithEvent3 extends JFrame implements ActionListener {
    // Attributes
    private JButton button = new JButton("Click Me");
    private int clickCount = 0;

    public GUIWithEvent3() {
        // Set the frame title
        super("GUI with Event");

        // Get the content pane and add a button to it
        getContentPane().add(button, BorderLayout.CENTER);

        // Register the event handler object to be the event handler
        // of the button
        button.addActionListener(this);

        // Set the default operation when the close button is
        // clicked
        setDefaultCloseOperation(EXIT_ON_CLOSE);

        // Set the GUI to the optimal size
        pack();
    }

    // Event handler method mandated by the ActionListener interface
    // for handling Action category event
    public void actionPerformed(ActionEvent ae) {
        clickCount++;
        button.setText("Clicked : " + clickCount);
    }
}
```

Figure 9.67 GUIWithEvent3.java

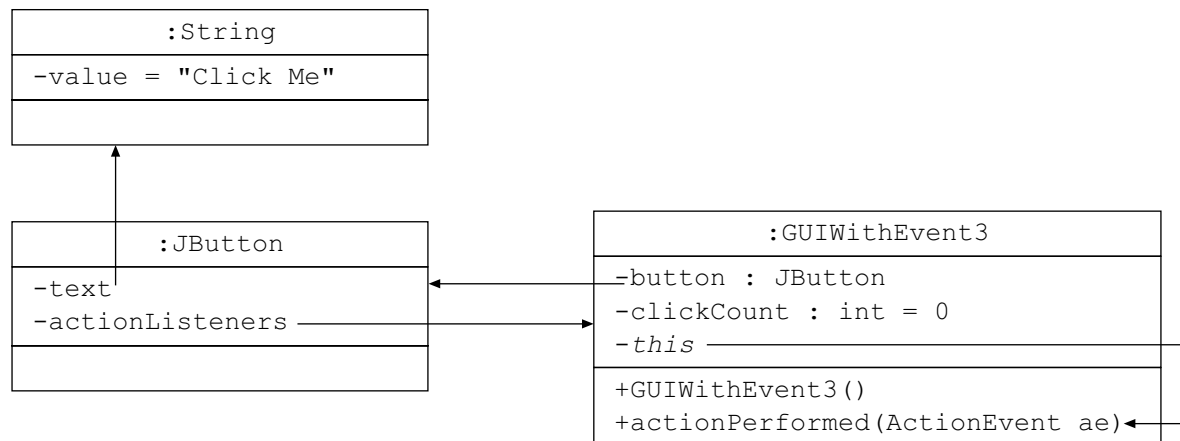
According to the definition of the `GUIWithEvent3` class

```
public class GUIWithEvent3 extends JFrame implements ActionListener {
    // Attributes
    private JButton button = new JButton("Click Me");
    ...
    public void actionPerformed(ActionEvent ae) {
        ...
    }
}
```

the interpretation is a `GUIWithEvent3` is a `JFrame`, has a `JButton` and plays the role of an `ActionListener`. As the class implements the interface `ActionListener`, it must define the `actionPerformed()` method, or the class becomes abstract. Please notice that the statement for registering an `ActionListener` object with the `JButton` object is:

```
button.addActionListener(this);
```

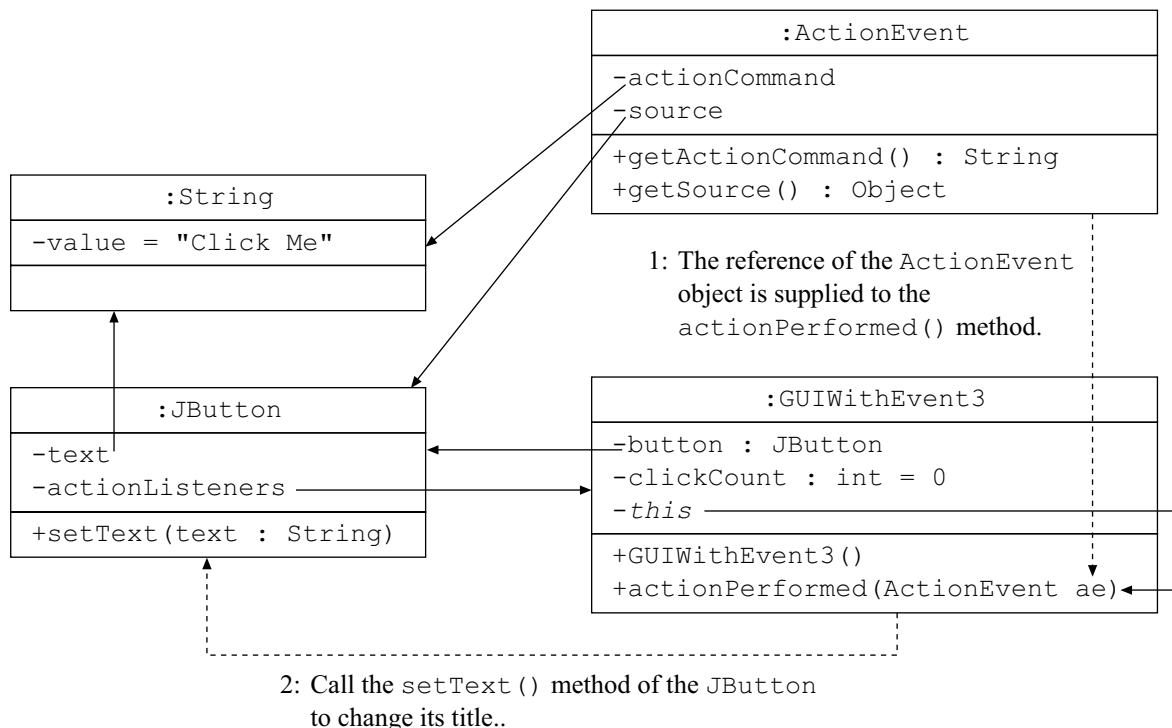
The statement is executed in the constructor of a `GUIWithEvent3` object. Therefore, the implicit variable `this` is referring to is the `GUIWithEvent` object. Furthermore, the `addActionListener()` method expects the supplied supplementary data to be an `ActionListener` object, which must be an object of its subclass. As the class `GUIWithEvent3` implements the interface `ActionListener`, a `GUIWithEvent3` object can also be considered to be an `ActionListener` object and can be supplied to the `addActionListener()` method. The scenario can be visualized in Figure 9.68.



**Figure 9.68** The relationships among the objects after a `GUIWithEvent3` object is created

This is similar to Figure 9.63, except the event handler is now a `GUIWithEvent3` object and not a dedicated event-handling object. (The implicit instance variable `this` is shown and is set in italics to highlight that when the `GUIWithEvent3` constructor is executing, the implicit instance variable `this` is referring to the `GUIWithEvent3` object itself.)

When the button of the **GUIWithEvent3** is clicked, an **ActionEvent** object is created. The source and action commands are referring to the **JButton** object and the text on the button respectively, as shown in Figure 9.69.



**Figure 9.69** The object involved and the sequence of operations when the **JButton** is clicked

This diagram is similar to Figure 9.66, except the event handler is now a **GUIWithEvent3** object and contains an **actionPerformed()** method. A driver program **TestGUIWithEvent3** is written and is stored on the course CD-ROM and website.

Please use the following self-test to test your understanding of handling Action category events.

### Self-test 9.5

Based on the classes that were written for editing the data of a clerk, a salesperson, and a manager respectively in question 3 of Self-test 9.4, equip the dialogs with the ability to show the contents of the text fields in the dialog when the <Edit> button is clicked. For example, if the <Edit> button of the manager dialog is clicked, the following output is shown in the Command Prompt:

```

Manager information
Name: [John]
Basic salary: [10000]
Allowance: [20000]
  
```

Furthermore, if the <Cancel> button is clicked, the dialog is hidden.

Based on the above requirements, which of the classes need modification, and what are the changes required?

---

The GUIs developed in this section are simple, for illustration. A more complicated GUI example, `SimpleCalculator1`, is provided in Appendix G. It has several `JButton`, `JTextField` and `JLabel` objects. How to handle events of such a GUI is discussed.

# Java applets

When Java programming was introduced in a public demonstration for the first time in 1995, two Java applets were shown that greatly impressed the audiences and brought about the popularity of the Java programming language. The reason is that Web pages could contain only *static* texts and images until that time, but the two applets *moved*. Later in the same year, it was announced that the Web browser, Netscape Navigator, would incorporate the Java technology that enabled all users in the world to enjoy the power of Java applets.

## What are Java applets?

Applets are classes written in the Java programming language. Similar to an image specified in a Web page, if a Web page specifies an applet, the browser downloads the class from the Web server and starts a Java virtual machine (JVM) for executing the downloaded class. If you have installed the Java SDK to your machine, the Web browser of your machine was configured for starting a JVM for executing Java applets.

As an applet is written in the Java programming language, it can accept user entries and process them according to the steps written in the program, and output the result to the user. When a Web page with a Java applet is shown on the screen, a region on the page is reserved for the Java applet. You can consider it as a panel or canvas in which GUI components can be placed or where the output can be drawn graphically on it.

There are several advantages of writing software as Java applets:

- 1 As applets are written in the Java programming language, which is cross-platform, applets embedded in Web pages can be executed by various platforms. For example, the same applet can be executed on a machine running a version of Microsoft Windows or UNIX without modifying a single line of program.
- 2 Applets can use the same rich standard software library as usual GUI applications written in the Java programming language.
- 3 Applet classes are downloaded automatically to the client machines whenever the Web browsers browse the Web page that embeds an applet. Therefore, no software installations are required. This feature is especially significant if the Java applet is to be executed by all staff members in a large company or by public users from all over the world.
- 4 The JVM that is started to execute Java applets implements special measures to prevent the Java applets from performing operations that may violate system security, such as accessing the files stored on the computer drives.



In the following subsections, we discuss how to write Java applets and embed them in Web pages to be viewed by Web browsers.

## Developing a simple Java applet

Java applets are downloaded to user Web browsers as compiled class files. Therefore, it is necessary to write the applet, place the compiled class file on the Web server after compilation and specify the class name in the Web page. To write a Java applet, you have to write a class that is a subclass of `javax.swing.JApplet` (or simply `JApplet`).

A `JApplet` object is a `JFrame`-like container to which GUI components can be added. You can apply the knowledge of building GUI that you learned earlier in this unit to construct a Java applet. One difference is that you usually use the `pack()` method of `JFrame` to set the GUI components to their optimal sizes, but the size of an applet in a Web page is specified in the Web page and the `JApplet` class therefore does not have the `pack()` method.

## Defining your own applet classes

To write a subclass of the `JApplet` class, the definition of an applet class should look like:

```
public class ClassName extends JApplet {
    // Attribute
    ...
    public ClassName() {
        // Add GUI components on it
        ...
    }
}
```

If the applet acts as an event listener object, the class should implement the suitable listener interface, such as:

```
public class ClassName extends JApplet implements Listener-interface-list {
    ...
}
```

An example class `HelloApplet` is written in Figure 9.70 to illustrate how to write a Java applet.

```

// Resolve classes in java.awt, java.awt.event, javax.swing package
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

// Definition of class HelloApplet
public class HelloApplet extends JApplet implements ActionListener {
    // Attribute
    private JButton button = new JButton("Please click me");

    // Constructor
    public HelloApplet() {
        // Get the content pane for adding component
        Container contentPane = getContentPane();

        // Add the JButton object to the content pane
        contentPane.add(button);

        // Register the HelloApplet object to be the Action category
        // event handler of the JButton
        button.addActionListener(this);
    }

    // Called if the JButton object is clicked
    public void actionPerformed(ActionEvent e) {
        JOptionPane.showMessageDialog(this, "Hello World");
    }
}

```

**Figure 9.70** HelloApplet.java

The constructor of a Java applet is similar to the constructor of the definition of a usual GUI class, except that there are no calls to the methods `setDefaultCloseOperation()` and `pack()`.

The constructor of the `HelloApplet` creates a `JButton` object to be added to its content pane. As the default layout manager of a `JApplet` is `BorderLayout`, the button is by default added to the center region of the content pane and the `JButton` object will occupy the entire region of the `JApplet` as all other regions contain no GUI components.

Whenever a Web browser reads a Web page that embeds a Java applet, it downloads the Java applet class on the fly, loads it into the JVM and creates an object for execution. Therefore, unlike a usual GUI program, it is not necessary to write a driver program to start a Java applet. Instead, you have to prepare a hypertext mark-up language (HTML) file that embeds the Java applet.

### Preparing the necessary HTML file

To view or activate a Java applet, you need a Web page or HTML file that embeds the Java applet. Web pages that can be viewed by a common Web browser written in HTML, so the extensions of the files are usually

either .htm or .html. The contents of an HTML file are plain texts so you can view or edit HTML files with a common editor such as Notepad. Furthermore, there are many software applications that can help you to edit a HTML file, such as Microsoft FrontPage.

To embed a Java applet in a HTML file so that a region on the Web page is dedicated to the applet, insert the following tab in the HTML file:

```
<applet code="AppletClassName" width="Width" height="Height">
</applet>
```

For example, the HTML file shown in Figure 9.71 embeds the Java applet HelloApplet.

```
<html>
<head>
<title>MT201 Unit 9</title>
</head>
<body>
<h1>HelloApplet</h1>
<applet code="HelloApplet" width="150" height="50">
</applet>
</body>
```

**Figure 9.71** hello.html

As the tags in an HTML file are case insensitive, all or some tags in the hello.html can be written in all capital letters. The three attributes — code, width and height of the applet tag — are mandatory for specifying the applet class, the width and height of the region dedicated to the applet respectively.

## Activating your applets

Placing the compiled class file HelloApplet.class, and the HTML file, hello.html in the same directory, you can test the applet using a tool that comes with the Java SDK — the appletviewer application.

Start a Command Prompt, assuming that you are using a version of Windows family, and change the current directory to the one that contains the class file and the HTML file. Then, you can execute the following command in the Command Prompt to test the applet

```
appletviewer HTML file
```

such as:

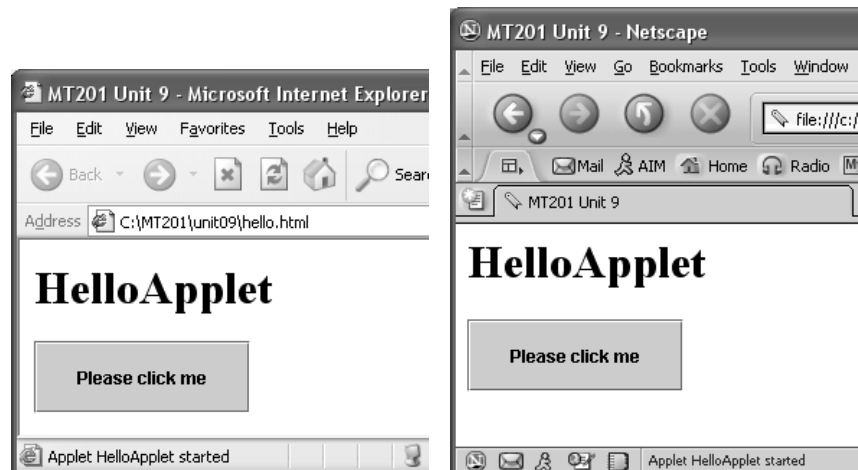
```
appletview hello.html
```

The appletviewer program reads the HTML file, loads the applet class file that is specified by the applet tag in the HTML file, creates an object of the applet class and executes it. Then, a window as shown in Figure 9.72 appears on the screen.



**Figure 9.72** Using the appletviewer tool to show a Java applet

As Java applets are targeted for execution by common Web browsers, you can use a Web browser to execute Java applets instead of using the appletviewer tool. For example, if you view the `hello.html` Web page with a Web browser such as Microsoft Internet Explorer or Netscape, the applet will be shown in a dedicated region on the Web page as shown in Figure 9.73.



**Figure 9.73** Viewing the HelloApplet with Microsoft Internet Explorer and Netscape

By comparing Figure 9.72 and Figure 9.73, you can see that the appletviewer program handles the applet tag only, whereas a common Web browser will show all contents provided in the HTML file. By clicking the button on the applet, a dialog box will be shown as in Figure 9.74.



By appletviewer



By a Web browser

**Figure 9.74** The dialogs shown by clicking the button on the HelloApplet

You should notice that no matter whether the applet is executed by `appletviewer` or a common Web browser, the dialog is shown with a status bar stating the dialog is a Java applet window. Such behaviour can help you to distinguish the windows opened by Java applet from those opened by native window software applications.

## The applet life cycle

In the previous sections, we discussed how to write Java applets and embed them in Web pages to be shown in a Web browser. Compared with standalone GUI programs that we discussed earlier in this unit, you should notice that their life cycles, which are the changes in their states during their existences, are different.

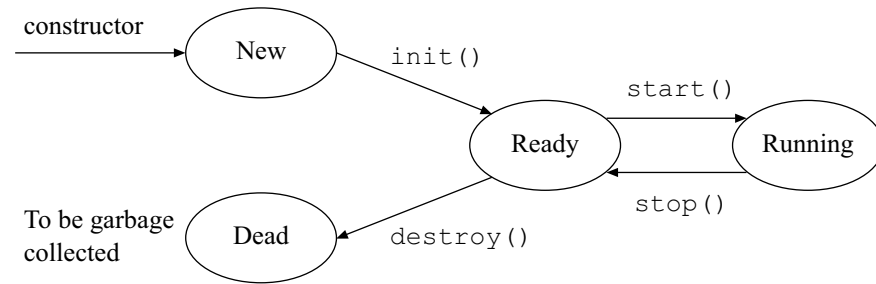
For a standard GUI program, an object of the class that models the GUI is created (and the constructor is executed), and either a `show()` method or `setVisible()` method with a supplementary data value of `true` is called to show the dialog on the screen. If the default close operation of the window is set to be `EXIT_ON_CLOSE`, clicking the <Close> button of the window will close the window and terminate the program. Otherwise, it is possible to register a `WindowListener` object with the GUI object that calls the `exit()` method in the `windowClosing()` method.

Conversely for a Java applet, the Web browser loads and creates an instance based on the downloaded applet class implicitly, and its constructor is therefore executed. The `java.awt.Applet` class, which is the superclass of the `javax.swing.JApplet`, defines some methods to be executed by the JVM of the Web browser in different situations.

**Table 9.4** The methods of an applet and the circumstances in which they are called

Methods of an applet to be called	Circumstances
<code>public void init()</code>	Called after the applet object is created. It is usually used to acquire applet-specific resources, such as getting images from the Internet.
<code>public void start()</code>	Called whenever the applet is about to be shown, such as the applet is shown on the screen for the first time and then reappears on the screen. A possible use of the method is to start the operations or set the state of the applet object while the applet is appearing on the screen.
<code>public void stop()</code>	Called whenever the applet is about to be hidden, such as the <Back> or <Front> button of the browser is clicked. This method can be used to perform some counter-operations with respect to the <code>start()</code> method.
<code>Public void destroy()</code>	Called whenever the applet is to be removed from the browser JVM. It is usually used for releasing the resources allocated during executing the <code>init()</code> method.

The methods mentioned in Table 9.4 can be summarized in the state diagram shown in Figure 9.75.



**Figure 9.75** The Java applet life cycle

Each oval in Figure 9.75 indicates a possible state of an applet throughout its life. When the state of an applet is changed from one state to another, the method associated with the arrow is called.

To illustrate the sequence of the applet methods to be called, an applet DemoApplet is written in Figure 9.76.

```

// Resolve the JApplet class
import javax.swing.*;

// Definition of class DemoApplet
public class DemoApplet extends JApplet {

    // Constructor
    public DemoApplet() {
        // Show a JLabel on the applet
        getContentPane().add(new JLabel("Hello World", JLabel.CENTER));
        // Show a message
        System.out.println("DemoApplet.DemoApplet()");
    }

    // Method inherited from java.awt.Applet to be called
    // under different situations

    public void destroy() {
        System.out.println("DemoApplet.destroy()");
    }

    public void init() {
        System.out.println("DemoApplet.init()");
    }

    public void start() {
        System.out.println("DemoApplet.start()");
    }

    public void stop() {
        System.out.println("DemoApplet.stop()");
    }
}

```

**Figure 9.76** DemoApplet.java

To show the `DemoApplet` applet, a Web page is required to specify the applet. A sample Web page, `demo.html`, is written and is available from the course CD-ROM and website. Use the `appletviewer` program, to view it, that is:

```
appletviewer demo.html
```

The `appletviewer` window appears on the screen as shown in Figure 9.77.



**Figure 9.77** The `DemoApplet` applet is shown with the `appletviewer` program

At the same time, the `String` objects sent to the standard output stream are shown in the Command Prompt

```
DemoApplet.DemoApplet()  
DemoApplet.init()  
DemoApplet.start()
```

confirming a `DemoApplet` object is created, and the `init()` and `start()` methods are called subsequently. If you minimize the `appletviewer` window, the `stop()` method is executed. The `start()` method will be executed again when the `appletviewer` window is restored. Finally, if you close the `appletviewer` window by clicking the <Close> button, the following message is shown on the screen:

```
DemoApplet.stop()  
DemoApplet.destroy()
```

The applet object is then garbage collected and the resources the applet object acquired, such as memory, are released.

If you view the `demo.html` with a common Web browser, there is no corresponding Command Prompt for showing the `String` objects sent to the standard output stream. To observe the message sent to the standard output stream by an applet object with a Web browser, please refer to discussion on the issue provided in Appendix I at the end of the unit.

## Summary

A software application with a user-friendly graphical user interface (GUI) can enable the users to use the software more easily and shorten the time to learn how to use the software. Before the Java programming language was released, the techniques for building graphical user interfaces were different for different platforms, which meant that the same applications could not migrate from one platform to another. As cross-platform technology, Java technology provides a single approach to build GUI applications so that the same GUI application can be executed on other platforms without any modification.

A critical problem of building cross-platform GUI applications is that the size of some GUI components are different on different platforms, and hence absolute positioning of the GUI components on the GUI cannot be adopted. To resolve the issue, GUIs built with Java technology use relative positioning by delegating the tasks to a layout manager of the container, so that the layout manager object rearranges the GUI components accordingly at runtime.

By using different layout managers with different policies, GUI components are arranged properly on the GUI. The responsibility of the program developer is to choose a suitable layout manager and settings. In the unit, we discussed three frequently used layout managers: `FlowLayout`, `BorderLayout` and `GridLayout`. For complicated GUIs, we can use a panel (modelled by `JPanel`), which can be treated as a GUI component but can contain other GUI components and can choose its own layout manager.

You can write a subclass of the `JFrame` class as the main window of the GUI. Its constructor is usually used to construct the layout. The content pane of a `JFrame` (obtainable by calling the `getContentPane()` method) is a container into which other GUI components can be added so that they are visible when the container is shown on the screen. It defines a `pack()` method that is used to set it and all GUI components added to it to their optimal sizes. Furthermore, the `setDefaultCloseOperation()` method can be used to set the operation when the window is to be closed, such as the `<Close>` button is clicked. For example, supplying the value specified by the final variable `EXIT_ON_CLOSE` to the method, clicking the `<Close>` button of the window would terminate the program.

On a GUI, we need some GUI components to accept user inputs and selections so that the program can operate accordingly. We discussed many GUI components that can be used to build a GUI.

To make a GUI responsive, you have to equip it with the ability to handle user interactions. The user interactions, such as clicking a button, are considered events. For a single user interaction with the GUI, such as clicking a button with the mouse pointer, the button is considered the event source, and an event object that encapsulates all information about the event is created. Then, if a suitable event handler (listener) object is



registered with the event source object, the corresponding method of the event handler object is called with the event object as the supplementary data, so that the method can determine what event occurred and respond accordingly.

You need the knowledge of Java interface to write event-handler classes, and hence event-handler objects can be created. Java interface is a variation of an abstract class discussed in *Unit 7*. All methods defined by a Java interface are `abstract` and the keyword `abstract` is therefore understood. As it provides no concrete method definitions but defines the method names and their parameter lists, and all its subclasses must define all the methods, Java interface can be used to mandate the methods to be defined by its subclasses.

With respect to the Java programming language, a class can extend a single class but can implement more than one interface. To write a subclass of an interface, the class is defined with an `implements` clause that specifies the interface(s) it implements, and the class must define all methods defined by the interface(s).

All possible events for a GUI are categorized, and a Java interface is defined for each event category. Therefore, the way to handle a particular event category is to write a class that implements the corresponding interface for that category and defines all methods specified by the interface. In the unit, we discussed a common event category, the Action category. Two other event categories are provided in Appendix F.

The Action category event is applicable to most GUI components and corresponds to the typical operation of a GUI component. For example, clicking a button is considered an Action category event to the button. It is therefore possible to register an Action category event-handler object with the button so that the corresponding method, the `actionPerformed()` method in this case, is called whenever the button is clicked.

Another common event category is the Window event. Operations such as clicking the <Close> button of the window and minimizing or maximizing the window are classified as a window event. For a GUI application, it is usually necessary to handle the event when the window is about to close, such as the user clicks the <Close> button, by prompting the user whether to save the unsaved data before terminating. To implement such a behaviour, you can write a class that implements the `WindowListener` interface and defines all methods specified by the interface. The `windowClosing()` method is called when the window is about to be closed.

Java applets are usually small Java classes and, with their GUIs, they can be embedded in a Web page to be shown by a Web browser. As an applet is written in the Java programming language and can use the same software library a normal GUI application uses, Java applets can basically perform whatever a normal GUI application can perform. However, as Java applets are downloaded from the Internet, it is unwise to execute Java applets without caution. As a result, the JVM of a Web

browser usually imposes some restrictions on the Java applets so that some operations are restricted, such as accessing the file stored on the computer drive.

The steps in writing and deploying Java applets are:

- 1 Write the applet class as the subclass of the `JApplet` class.
- 2 Compile the applet class definition and place the compiled class file on a Web server.
- 3 Write a Web page that specifies the Java applet with an `applet` tag.
- 4 Enable Web browser.

## Appendix A: Using labels for showing operation results

Besides showing static text, a label is used for showing the operation results, such as the display of a calculator as shown in Figure 9.2. To change the text of a JLabel, similar to a JLabel object, you can call its `setText()` method. For example, the `FrameWithLabel1` class is modified to be `FrameWithLabel2` class as shown in Figure 9.78 that defines a new method `countDown()`.

```
// Resolve class Container
import java.awt.*;
// Resolve class JFrame and JLabel
import javax.swing.*;

// Definition of class FrameWithLabel2
public class FrameWithLabel2 extends JFrame {
    // The prefix of the message to be shown
    private static final String MESSAGE = "Count Down : ";

    // Attribute
    private JLabel label;    // The label that the frame possesses

    // Constructor
    public FrameWithLabel2() {
        // Call super class constructor with a title
        super("Frame With label");

        // Create a JLabel
        label = new JLabel(MESSAGE);

        // Get the content pane
        Container contentPane = getContentPane();

        // Add the JLabel object to the content pane
        contentPane.add(label);

        // Set when the close button is clicked, the application exits
        setDefaultCloseOperation(EXIT_ON_CLOSE);

        // Reorganize the embedded components
        pack();
    }

    // Count down and the values are shown on the label
    public void countDown(int limit) {
        for (int i=limit; i >= 0; i--) {
            label.setText(MESSAGE + i);
        }
    }
}
```

**Figure 9.78** `FrameWithLabel2.java`

A driver program, the `TestFrameWithLabel2` class, is written to test the `FrameWithLabel2` class. Its definition is shown in Figure 9.79.

```
// Definition of class TestFrameWithLabel2
public class TestFrameWithLabel2 {

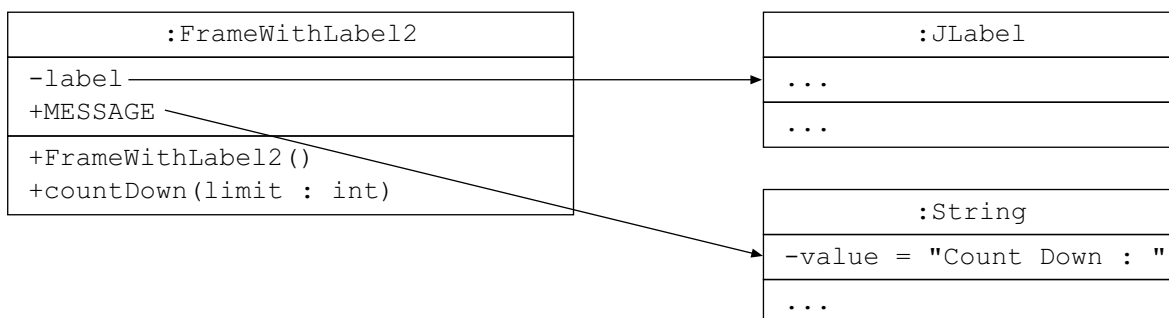
    // Main executive method
    public static void main(String args[]) {
        // Create a FrameWithLabel2 object
        FrameWithLabel2 frame = new FrameWithLabel2();

        // Show it on the screen
        frame.show();

        // Count down from 10000
        frame.countDown(10000);
    }
}
```

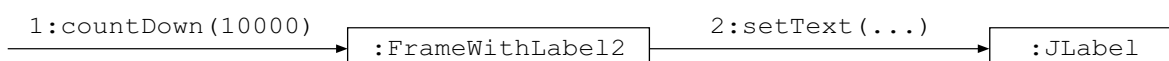
**Figure 9.79** `TestFrameWithLabel2.java`

While executing the constructor of a `FrameWithLabel2` object, a `JLabel` object is created (referred by the attribute `label` of the `FrameWithLabel2` object) as shown in Figure 9.80.



**Figure 9.80** The scenario after a `FrameWithLabel2` object is created

After the `FrameWithLabel2` is shown on the screen by calling its `show()` method, the `TestFrameWithLabel2` class `main()` method sends a message `countDown` with supplementary data of 10000 to the `FrameWithLabel2` object referred by variable `frame`. Then, the `FrameWithLabel2` object repeatedly sends message `setText` with supplementary data of the reference of a `String` object to the `JLabel` object referred by the attribute `label` so that the title of the `JLabel` object is changed accordingly. The sequence of message passing is visualized in Figure 9.81.



**Figure 9.81** The sequence of message passing for a method call of `countDown(10000)`

Compile the classes `FrameWithLabel2` and `TestFrameWithLabel2` and execute the `TestFrameWithLabel2`. A `FrameWithLabel2` object is shown on the screen in which the text of the `JLabel` changes rapidly until the GUI becomes Like Figure 9.82.



**Figure 9.82** The `FrameWithLabel2` object after executing the `countDown()` method

## Appendix B: Further discussion of the three layout manager classes

Building GUIs in Java using layout managers enables you to consider the allocation of GUI components in a logical way. The GUI components are allocated physically at runtime so that you can have a properly built GUI. Also, the users may change the size of the frame and hence the size of the container (such as the content pane of a `JFrame` object). Resizing a container that uses a specific layout manager will change the sizes and/or locations of the embedded GUI components. Therefore, it is necessary to consider such behaviour so that the GUI components are still properly allocated.

### Resizing behaviours of different layout managers

#### `FlowLayout` layout manager

Resize the window and you will find the characteristics of a `Container` with `FlowLayout` object as its layout manager. Enlarging the window of the `FrameWithFlowLayout` object as shown in Figure 9.83, the GUI becomes:



**Figure 9.83** The `FrameWithFlowLayout` object after enlarging

Figure 9.83 shows that the sizes of the `JButton` objects are kept unchanged. Furthermore, they are aligned to the center and placed at the top of the GUI.

Then, reducing the width of the window, the GUI becomes as shown in Figure 9.84.



**Figure 9.84** The width of the `FrameWithFlowLayout` object is reduced

The `FlowLayout` layout manager arranges the GUI components as if they are words entered into a word-processor software application and the default alignment is center. If the line has sufficient space, all words are aligned center in a single line. If the margins of the page are changed so that line width is reduced, the words that exceed the line width will be wrapped to the next line. At any time, the widths of all words are kept unchanged.

According to the above arrangement rules, if the width of the GUI as shown in Figure 9.85 is further reduced, the GUI will become as shown in Figure 9.85.

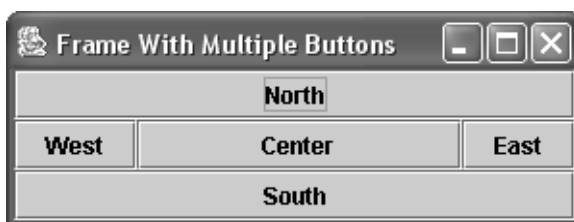


**Figure 9.85** The width of the `FrameWithFlowLayout` object is further reduced

Figure 9.85 shows that the widths of the `JButton` objects are different because the length of their titles are different. Their preferred sizes are shown in a `Container` with the `FlowLayout` object as layout manager.

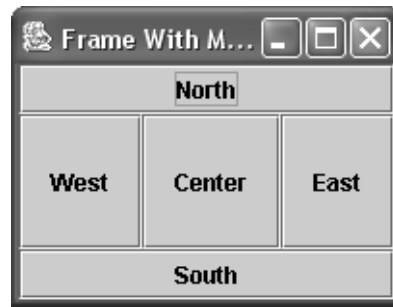
### **BorderLayout layout manager**

We can now further investigate a GUI with a `BorderLayout` object as its layout manager. Each GUI component is resized to occupy the entire region. For example, the width of the `JButton` objects in the south and north regions are set to fill the entire regions. If you widen the window of the `FrameWithBorderLayout` object, the GUI will become as shown in Figure 9.86.



**Figure 9.86** A widened `FrameWithBorderLayout` object

If the width of the `Container` object is extended, the width of the GUI components at the north, center, and south regions are extended to fill the extra spaces. If the size of the `Container` object is extended vertically, however, the height of the GUI components at the east, center, and west regions are extended as well to fill the extra spaces, such as the `FrameWithBorderLayout` shown in Figure 9.87.



**Figure 9.87** A `FrameWithBorderLayout` object with window height increased

You can see that the embedded GUI components are always resized so that they occupy the entire space of the `Container` object.

As mentioned, all five regions are optional. If a region is missing, the remaining regions usually occupy the space properly so that the entire GUI components are completely filled. However, you should notice that if the center region is missing, there might be unused space in the `Container` object. Therefore, it is preferable to use the center region whenever possible.

To illustrate a `Container` object with different combinations of the regions, a class `FrameWithBorderLayout2` is written in Figure 9.88 that can add `JButton` objects to the regions according to the value passed to its constructor.

```
// Resolve class Container
import java.awt.*;
// Resolve class JFrame and JButton
import javax.swing.*;

// Definition of class FrameWithBorderLayout2
public class FrameWithBorderLayout2 extends JFrame {
    // Attribute
    private JButton buttonEast;    // The east button
    private JButton buttonSouth;   // The south button
    private JButton buttonWest;    // The west button
    private JButton buttonNorth;   // The north button
    private JButton buttonCenter;  // The center button

    // Constructor
    public FrameWithBorderLayout2(int combination) {
        // Call super class constructor with a title
        super("Frame With Multiple Buttons");

        // Create JButton objects
        buttonEast = new JButton("East");
        buttonSouth = new JButton("South");
        buttonWest = new JButton("West");
        buttonNorth = new JButton("North");
        buttonCenter = new JButton("Center");

        // Get the content pane
        Container contentPane = getContentPane();
```



```

// Add the JButton objects to the content pane
if ((combination & 1) != 0) {
    contentPane.add(buttonEast, BorderLayout.EAST);
}
if ((combination & 2) != 0) {
    contentPane.add(buttonSouth, BorderLayout.SOUTH);
}
if ((combination & 4) != 0) {
    contentPane.add(buttonWest, BorderLayout.WEST);
}
if ((combination & 8) != 0) {
    contentPane.add(buttonNorth, BorderLayout.NORTH);
}
if ((combination & 16) != 0) {
    contentPane.add(buttonCenter, BorderLayout.CENTER);
}

// Set when the close button is clicked, the application exits
setDefaultCloseOperation(EXIT_ON_CLOSE);

// Reorganize the embedded components
pack();
}
}

```

**Figure 9.88** FrameWithBorderLayout2.java

The difference between the definitions of classes

FrameWithBorderLayout and FrameWithBorderLayout2 is that its constructor accepts a parameter of type `int` that controls the regions to be added to the content pane. The `add()` method for a particular region of the content pane is called if the condition of the `if` statement that looks like the following is true.

```

if ((combination & 1) != 0) {
    contentPane.add(buttonEast, BorderLayout.EAST);
}

```

The sequence of `if` statements borrows the idea of binary numbers that each bit of a number controls whether a `JButton` object is added to a particular region.

centre	north	west	south	east
<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Therefore, in order to add all five regions to the `Container` object, a value of `int` that is equivalent to binary number of  $11111_2$  (that is, 31 in the decimal number system) must be supplied to the constructor.

To test the `FrameWithBorderLayout2`, a class

`TestFrameWithBorderLayout2` is written in Figure 9.89 that converts the first program parameter to a value of type `int` and supplies the value to the constructor of the `FrameWithBorderLayout2` object.

```
// Definition of class TestFrameWithBorderLayout2
public class TestFrameWithBorderLayout2 {

    // Main executive method
    public static void main(String args[]) {
        if (args.length == 0) {
            System.out.println(
                "Usage : java FrameWithBorderLayout2 " +
                "<region combinations>");
        }
        else {
            // Get the integer value from the program parameter
            int regions = Integer.parseInt(args[0]);

            // Create a FrameWithBorderLayout2 object with the specified
            // regions
            FrameWithBorderLayout2 frame =
                new FrameWithBorderLayout2(regions);

            // Show it on the screen
            frame.show();
        }
    }
}
```

**Figure 9.89** TestFrameWithBorderLayout2.java

Compile the classes and execute the `TestFrameWithBorderLayout2` class with a program parameter of value ranging from 1 to 31. You will get a different combination of the regions shown on the screen. For example, executing the program with a program parameter of 26, the GUI as shown in Figure 9.90 is shown on the screen.



**Figure 9.90** The GUI to be shown by providing a program parameter of value 26

As mentioned, if the center region is missing, there may be unused space in the GUI. For example, Figure 9.91 shows a GUI that uses the north, east, and south regions.



**Figure 9.91** The GUI with `BorderLayout` layout manager that uses the north, east and south regions only

You should remind yourself that if you need a GUI with three regions in a column, you should choose the north, center and south regions instead of the north, east (or west) and south regions as in Figure 9.90.

### GridLayout layout manager

If you alter the width and/or the height of a GUI that uses a `GridLayout` object as its layout manager, and the GUI components are arranged in three rows and two columns, its size is adjusted and it looks like the one shown in Figure 9.92.



**Figure 9.92** A resized `FrameWithGridLayout` object

You can see that all embedded GUI components of a `Container` are resized properly so that all the extra spaces are fully occupied and are shared among all. Therefore, the sizes of all embedded GUI components are equal.

## Other properties of layout managers

### The alignment property of `FlowLayout` layout manager

We said that if the GUI components do not occupy the entire row in the container, they are by default aligned to the center. It is actually possible to change such alignment behaviour, which is usually set while creating the `FlowLayout` object. The `FlowLayout` class defines an overloaded constructor

```
public FlowLayout(int align)
```

that accepts supplementary data of type `int` for the alignment property. The class also defines the `public final` class variable for different alignments, including `CENTER`, `LEFT` and `RIGHT`. For example, to create a `FlowLayout` object that aligns GUI components to the right, the statement is:

```
new FlowLayout(FlowLayout.RIGHT)
```

## The gaps among GUI components

The components of all GUIs built in this unit so far are closely packed together. However, the GUIs usually look nicer if the components are arranged with gaps among them. All layout manager classes define the vertical gap and horizontal gap properties, which are usually set by supplying supplementary data to the constructor. The constructors that accept supplementary data for vertical and horizontal gaps are:

```
public FlowLayout(int align, int hgap, int vgap)
public BorderLayout(int hgap, int vgap)
public GridLayout(int rows, int cols, int hgap, int vgap)
```

The parameters `hgap` and `vgap` that are accepted by the above three constructors specify the horizontal gap and vertical gap in pixels respectively.

## Appendix C: On the use of JOptionPane

Calling the class methods of the `JOptionPane` class can show three different standard dialogs. The methods are:

```
public static void showConfirmDialog(...)
public static void showInputDialog(...)
public static void showMessageDialog(...)
```

The above three methods are overloaded, and each overloaded version accepts a different number of supplementary data. The following is a list of commonly used ones:

```
public static int showConfirmDialog(
    Component parentComponent,
    Object message)
public static int showConfirmDialog(
    Component parentComponent,
    Object message,
    String title,
    int optionType)
public static int showConfirmDialog(
    Component parentComponent,
    Object message,
    String title,
    int optionType,
    int messageType)

public static String showInputDialog(Object message)
public static String showInputDialog(
    Object message,
    Object initialSelectionValue)
public static String showInputDialog(
    Component parentComponent,
    Object message)
public static String showInputDialog(
    Component parentComponent,
    Object message,
    Object initialSelectionValue)
public static String showInputDialog(
    Component parentComponent,
    Object message,
    String title,
    int messageType)

public static void showMessageDialog(
    Component parentComponent,
    Object message)
public static void showMessageDialog(
    Component parentComponent,
    Object message,
    String title,
    int messageType)
```

Different methods accept a different set of supplementary data. You can first determine the type of dialog to be shown and then select the one that can accept all settings for the dialog to be shown. The effects of the different parameters are presented in Table 9.5.

**Table 9.5** The effects of different supplementary data to be supplied for showing dialogs

Parameter	Effect
<code>parentComponent</code>	The parent component of the dialog, which can be a <code>JFrame</code> object or any GUI component, is used for determining the position of the dialog according to the position of the parent component. If a value of null is supplied, the dialog is usually positioned at the center of the screen.
<code>message</code>	The reference of a <code>String</code> is usually supplied and is shown in the dialog as the message.
<code>messageType</code>	The type of message to be shown by the dialog. Usually, a default icon is shown for different message types. Possible message types are represented by public final class variables, <code>ERROR_MESSAGE</code> , <code>INFORMATION_MESSAGE</code> , <code>WARNING_MESSAGE</code> , <code>QUESTION_MESSAGE</code> and <code>PLAIN_MESSAGE</code> .
<code>title</code>	The title of the dialog.
<code>optionType</code>	These supplementary data are applicable to the <code>showConfirmDialog()</code> method only. They specify the option buttons to be shown in the dialog. Possible values to be supplied are represented by public final class variables, <code>DEFAULT_OPTION</code> (showing the default buttons for different message types), <code>YES_NO_OPTION</code> (showing the Yes and No buttons), <code>YES_NO_CANCEL_OPTION</code> (showing the Yes, No and Cancel buttons), <code>OK_CANCEL_OPTION</code> (showing the OK and Cancel buttons).
<code>initialSelectionValue</code>	These supplementary data are applicable to the <code>showInputDialog()</code> method only. They specify the initial contents of the input field on the dialog.

The `showConfirmDialog()` methods show a dialog on the screen with a message and some buttons specified by the `message` and `optionType` parameters. Once a user clicks a button on the dialog, the dialog is hidden and a value of type `int` is returned indicating the button clicked. Possible return values are, `CANCEL_OPTION`, `NO_OPTION`, `OK_OPTION`, `YES_OPTION` if the <Cancel>, <No>, <OK> and <Yes> buttons are clicked respectively.

The `showInputDialog()` methods show a dialog on the screen with a message and an input field for the user to enter. If there is default value to be provided to the user, it can be supplied to the method via the `initialSelectionValue` parameter. If the user clicks the <OK>

button to complete the dialog, the reference of a `String` object is returned with contents of the input field. Otherwise, a value of `null` is returned if the user clicked the `<Cancel>` button.

The `showMessageDialog()` methods are only used to show a message dialog on the screen, and the user clicks the `<OK>` button to complete it and no value is to be returned.

## Appendix D: Dialogs

Dialogs are a special type of frame that are shown on the screen for either getting user input or notifying the user with a particular message that needs the user's immediate attention. Usually, when a dialog is shown on the screen, the original frame becomes unresponsive until the user closes the dialog. Such a dialog is considered to be modal. If both the original frame and the dialog are active at the same time, the dialog is known as modeless.

As dialogs are frequently used GUI widgets, especially standard input and message dialogs, the Swing packages provide the `javax.swing.JOptionPane` class (or simply the `JOptionPane` class) for creating and using the standard dialogs, which you have been using since *Unit 4*. Basically, there are three standard dialogs: message dialog, input dialog and confirmation dialog. A class `TestDialogs` is written as shown in Figure 9.93 to illustrate a possible use of the three dialogs.

```
// Resolve JOptionPane class
import javax.swing.JOptionPane;

// Definition of class TestDialogs
public class TestDialogs {

    // Main executive method
    public static void main(String args[]) {
        int userInput;
        do {
            // Show the message dialog as introduction
            JOptionPane.showMessageDialog(null,
                "This program illustrates the uses of dialogs");

            // Show a dialog to get the username from the user
            String username;
            do {
                username = JOptionPane.showInputDialog(null,
                    "What is your name?");
            } while (username == null);

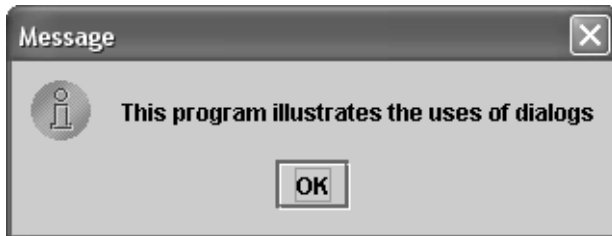
            // Prompt user whether to continue execution
            userInput =
                JOptionPane.showConfirmDialog(
                    null,
                    "Hello, " + username +
                    "\nDo you want to continue?",
                    "TestDialogs",
                    JOptionPane.YES_NO_OPTION);
        }
        while (userInput == JOptionPane.YES_OPTION);

        // Terminate the program explicitly
        System.exit(0);
    }
}
```

Figure 9.93 TestDialogs.java



The `JOptionPane` class defines overloaded `showMessageDialog()`, `showInputDialog()` and `showConfirmDialog()` for showing the three standard dialogs on the screen. Compile and execute the `TestDialogs` class. It first shows a message dialog as shown in Figure 9.94 on the screen as the result of executing the class method `showMessageDialog()`.



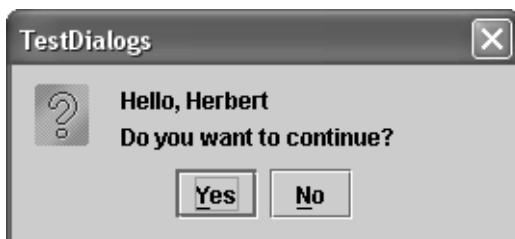
**Figure 9.94** The message dialog shown by calling the class method `showMessageDialog()` of `JOptionPane` class

Then, click the <OK> button to complete the dialog. The class method `showInputDialog()` is executed. The next input dialog as shown in Figure 9.95 appears on the screen.



**Figure 9.95** The input dialog shown by calling the class method `showInputDialog()` of `JOptionPane` class

Input a name in the text field on the input dialog. Click the <OK> button to complete the dialog. A `String` object containing the entered name is returned as the return value of the `showInputDialog()` method. Finally, the class method `showConfirmDialog()` of the `JOptionPane` class is executed. A confirmation dialog as shown in Figure 9.96 appears on the screen.



**Figure 9.96** The confirmation dialog shown by calling the class method `showConfirmDialog()` of `JOptionPane` class

Clicking either the <Yes> or the <No> button closes the dialog. The difference is that the returned value of the `showConfirmDialog()` returns a value denoted by class variable `YES_OPTION` and `NO_OPTION`

of the `JOptionPane` class by clicking the <Yes> button and the <No> button respectively.

The `TestDialogs` class illustrates a possible way to use the three different standard dialogs prepared by the `JOptionPane` class. It is possible to customize the dialogs to be shown on the screen further, such as the dialog title. For a detailed discussion on the use of the `JOptionPane` class, please refer to Appendix C. Besides using the `JOptionPane` class to show standard dialogs, you can develop your own dialogs by writing subclasses of the `JDialog` class.

## User-defined dialog

Besides using the standard dialog provided by the `JOptionPane` class, you can develop your own dialogs by writing subclasses of the `JDialog` (`javax.swing.JDialog`) class. First, it is necessary to understand the `JDialog` class.

For creating a `JDialog` object, the following is a list of commonly used overloaded constructors:

```
public JDialog()
public JDialog(Frame owner)
public JDialog(Frame owner, String title)
public JDialog(Frame owner, boolean modal)
public JDialog(Frame owner, String title, boolean modal)
```

Although the list of constructors is quite confusing and you may not know which one to use to create a `JDialog` object, the five constructors actually set three different properties of a dialog as shown in Table 9.6.

**Table 9.6** The effects of the supplementary data supplied to the `JDialog` constructors

Property	Effects
owner	The frame that shows the dialog. The type is <code>java.awt.Frame</code> , which is a superclass of the <code>JFrame</code> class. It is therefore possible to pass the reference of a <code>JFrame</code> object for the owner parameter. If the frame does not show a dialog, a value of <code>null</code> can be supplied to the constructor.
Title	The dialog title.
modal	Determine whether the created dialog is modal or modeless by supplying the value <code>true</code> and <code>false</code> respectively. If the reference of a frame is supplied to the constructor as owner and a value of <code>true</code> is supplied as modal, the frame becomes unresponsive while the dialog is shown on the screen.

Like a `JFrame` object, a `JDialog` object embeds no GUI components. It is therefore necessary to add the GUI components to it one by one as if it

is a `JFrame` object. After a `JDialog` object is created, you can call its `show()` method or its `setVisible()` method with supplementary data of `true` to show it on the screen. However, you should notice that if the `show()` method or `setVisible()` method with supplementary data of `true` of a `JDialog` object is called, it does not return until the dialog is closed, such as clicking the <Close> button of the dialog, or is hidden by calling the `hide()` method or `setVisible()` method with supplementary data `false`.

A class `TestJDialog` is written in Figure 9.97 for testing with a `JDialog` object.

```
// Resolove classes in javax.swing package
import javax.swing.*;

// Definition of class TestJDialog
public class TestJDialog {

    public static void main(String args[]) {
        // Create a JFrame object
        JFrame frame = new JFrame("The main frame");
        // Create a JDialog object with the JFrame as its owner
        JDialog dialog = new JDialog(frame, "A dialog", true);

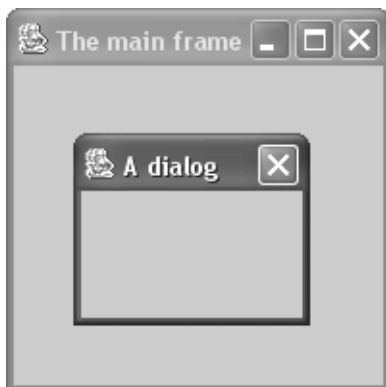
        // Explicitly set the sizes of the frame and the dialog
        // as they contain no GUI components
        frame.setSize(200, 200);
        dialog.setSize(100, 100);

        // Show the frame and then the dialog
        frame.show();
        dialog.show();

        // Terminate the program explicitly
        System.exit(0);
    }
}
```

**Figure 9.97** TestJDialog.java

Compile and execute the `TestJDialog` class. A frame and a dialog are shown on the screen as in Figure 9.98.



**Figure 9.98** A frame and a dialog are shown on the screen by the `TestJDialog` program

An observation from Figure 9.98 is that the appearances of a `JDialog` and a `JFrame` are different in that a `JDialog` object, by default, does not have the <Minimize> and <Maximize> button. While the `JDialog` and the `JFrame` are shown on the screen, you can interact with the `JDialog` object, such as resizing it or moving it on the screen. However, the `JFrame` object is unresponsive, and clicking the `JFrame` object may give you an alert sound if your computer is equipped with audio devices. This is what we mentioned as a modal dialog.

Furthermore, the `main()` method of the `TestJDialog` hints that executing the `show()` method of a `JFrame` object will return immediately once the `JFrame` object is shown on the screen, but calling the `show()` method of a `JDialog` object will not return until the dialog is closed. Otherwise, the `System.exit()` method would have been executed and the program would have terminated immediately. Once the <Close> button of the `JDialog` object is clicked, the dialog hides. The `show()` method of the `JDialog` object returns, and the flow of control returns to the `main()` method of the `TestJDialog` class. Then, the `System.exit()` method is executed, and the program terminates as a result.

Due to the behaviour of the `show()` method of a `JDialog` class, a user-defined dialog is usually written as shown in the following template:

```
public class DialogClass extends JDialog [implements EventListener(s)] {
    // GUI components
    ...
    // Attribute(s) to be return
    ...
    // Constructor
    public DialogClass(Frame owner, String title, boolean modal) {
        // Call the superclass constructor to set the attributes
        super(owner, title, modal);
        // Arrange the GUI components
        ...
        // Register event handlers
        ...
    }
    // Getter method(s) for the attribute(s)
    public getAttribute() {
    }
    // Event handler methods that manipulate the attribute(s)
    ...
}
```

The constructor builds the GUI, and event handlers are registered to the GUI components on the dialog. The event handler methods are designed to manipulate the attribute(s) of the dialog object. Whenever the dialog is closed, the getter methods of the dialog can be used to inquire the user selections. A sample program segment is:

```
DialogClass dialog = new DialogClass(...); // Create the dialog object
...
dialog.show(); // Show the dialog and handle the user interactions
selection = dialog.getAttribute(); // Get the user selection or entry
```

The case study of the payroll software provided in Appendix G of the unit uses several user-defined dialogs for getting user inputs. Please refer to that appendix for further examples of using user-defined dialogs.

## Appendix E: Further discussion of Java interfaces

This appendix discusses how to define and use Java interfaces. As mentioned, Java interface can only define public class variables and abstract methods. A Java interface `Time` can be defined as shown in Figure 9.99.

```
// Definition of interface Time
public interface Time {
    // Constant class variables
    public static int MAX_HOUR = 23;
    public static int MAX_MINUTE = 59;
    public static int MAX_SECOND = 59;
    // Getter methods
    public int getHour();
    public int getMinute();
    public int getSecond();
    // Setter methods
    public void setHour(int hour);
    public void setMinute(int minute);
    public void setSecond(int second);
    // Utility methods
    public void tick();
    public int findDifference(Time time);
}
```

**Figure 9.99** `Time.java`

The optional keywords `public` and `public static final` are usually added to method declarations and variable declarations for clarification. The above interface declares a type named `Time` with constant class variables `MAX_HOUR`, `MAX_MINUTE` and `MAX_SECOND`. Furthermore, it defines getter/setter methods and a method `tick()`.

You can see that an interface defines no constructor and no executable method body. Like an abstract class, it only defines the methods its subclasses have to implement. In *Unit 7*, we said that there are two different implementation approaches that use three attributes for hour, minute and second, and one attribute for `secondsSinceMidnight`.

To define subclasses of an interface, note that the keyword `implements` is used instead of `extends`. Based on the two approaches we discussed in *Unit 7*, two subclasses of the interface `Time` are written as shown in Figure 9.100 and Figure 9.101 respectively.

```
// Definition of class SimpleTime
public class SimpleTime implements Time {
    // Attributes
    private int hour;        // The hour of time
    private int minute;      // The minute of time
    private int second;      // the second of time

    public SimpleTime(int hour, int minute, int second) {
        this.hour = hour;
        this.minute = minute;
        this.second = second;
    }

    // Get the value of attribute hour
    public int getHour() {
        return hour;
    }

    // Get the value of attribute minute
    public int getMinute() {
        return minute;
    }

    // Get the value of attribute second
    public int getSecond() {
        return second;
    }

    // Set the value of attribute hour
    public void setHour(int theHour) {
        if (0 <= theHour && theHour <= MAX_HOUR) {
            hour = theHour;
        }
        else {
            System.out.println(
                "Invalid hour. Object attribute kept unchanged.");
        }
    }

    // Set the value of attribute minute
    public void setMinute(int theMinute) {
        if (0 <= theMinute && theMinute <= MAX_MINUTE) {
            minute = theMinute;
        }
        else {
            System.out.println(
                "Invalid minute. Object attribute kept unchanged.");
        }
    }
}
```

```

// Set the value of attribute second
public void setSecond(int theSecond) {
    if (0 <= theSecond && theSecond <= MAX_SECOND) {
        second = theSecond;
    }
    else {
        System.out.println(
            "Invalid second. Object attribute kept unchanged.");
    }
}

// Advance the time by one second
public void tick() {
    second++;
    if (second > MAX_SECOND) {
        second = 0;
        minute++;
    }
    if (minute > MAX_MINUTE) {
        minute = 0;
        hour++;
    }
    if (hour > MAX_HOUR) {
        hour = 0;
    }
}

// Show the time in the format hh:mm:ss on the screen
public String toString() {
    return
        ((hour < 10) ? "0" + hour : "" + hour) + ":" +
        ((minute < 10) ? "0" + minute : "" + minute) + ":" +
        ((second < 10) ? "0" + second : "" + second);
}
}

```

**Figure 9.100** SimpleTime.java

```

// Definition of class ConsolidatedTime
public class ConsolidatedTime implements Time {
    // Attributes
    private int secondSinceMidnight;

    public ConsolidatedTime(int hour, int minute, int second) {
        setHour(hour);
        setMinute(minute);
        setSecond(minute);
    }

    // Get the value of attribute hour
    public int getHour() {
        return secondSinceMidnight / 3600;
    }
}

```



```
// Get the value of attribute minute
public int getMinute() {
    return (secondSinceMidnight % 3600) / 60;
}

// Get the value of attribute second
public int getSecond() {
    return (secondSinceMidnight % 60);
}

// Set the value of attribute hour
public void setHour(int theHour) {
    if (0 <= theHour && theHour <= MAX_HOUR) {
        secondSinceMidnight =
            theHour * 3600 +
            (secondSinceMidnight % 3600);
    }
    else {
        System.out.println(
            "Invalid hour. Object attribute kept unchanged.");
    }
}

// Set the value of attribute minute
public void setMinute(int theMinute) {
    if (0 <= theMinute && theMinute <= MAX_MINUTE) {
        secondSinceMidnight =
            (secondSinceMidnight / 3600 * 3600) +
            theMinute * 60 +
            (secondSinceMidnight % 60);
    }
    else {
        System.out.println(
            "Invalid minute. Object attribute kept unchanged.");
    }
}

// Set the value of attribute second
public void setSecond(int theSecond) {
    if (0 <= theSecond && theSecond <= MAX_SECOND) {
        secondSinceMidnight =
            secondSinceMidnight / 60 * 60 +
            theSecond;
    }
    else {
        System.out.println(
            "Invalid second. Object attribute kept unchanged.");
    }
}

// Advance the time by one second
public void tick() {
    secondSinceMidnight = (secondSinceMidnight + 1) % 86400;
}
```

```

// Show the time in the format hh:mm:ss on the screen
public String toString() {
    int hour = getHour();
    int minute = getMinute();
    int second = getSecond();
    return
        ((hour < 10) ? "0" + hour : "" + hour) + ":" +
        ((minute < 10) ? "0" + minute : "" + minute) + ":" +
        ((second < 10) ? "0" + second : "" + second);
}
}

```

**Figure 9.101** ConsolidatedTime.java

The classes `SimpleTime` and `ConsolidatedTime` are subclasses of the interface `Time`. They define all abstract methods defined in the interface `Time` in their own way. Although the type `Time` is an interface without any concrete method, it can be used as the type of the variable, such as:

```
Time time1;
```

The type of the variable `time1` is non-primitive, which means that it can only store the reference of an object. Like an abstract class, it is not possible to create an object of such a type. Therefore, a variable of an interface type can only refer to an object of its concrete subclasses. For example, with the interface `Time` and the classes `SimpleTime` and `ConsolidatedTime`, the following program segment is possible:

```
Time time1 = new SimpleTime1();
Time time2 = new ConsolidatedTime1();
```

To test the classes, a driver program is written as shown in Figure 9.102.

```

// Definition of class TestTime
public class TestTime {

    // Main executive method
    public static void main(String args[]) {
        // Create a SimpleTime and a ConsolidatedTime object
        Time time1 = new SimpleTime(11, 59, 59);
        Time time2 = new ConsolidatedTime(23, 59, 59);
        // Increase the objects by one second
        time1.tick();
        time2.tick();
        // Show the objects on the screen
        System.out.println(time1);
        System.out.println(time2);
    }
}

```

**Figure 9.102** TestTime.java

In the `main()` method of the `TestTime` class, the variables `time1` and `time2` are of type `Time` and it is therefore possible to send messages specified by the interface `Time`. As the variable `time1` is referring to a `SimpleTime` object, the `SimpleTime` object receives a message and performs the operation according to the class definition. Similarly, messages sent to the `ConsolidatedTime` object via the variable `time2` perform the operations according to the `ConsolidatedTime` class definition.

As usual, the `SimpleTime` and `ConsolidatedTime` are defined without the `extends` clause. You can imagine that an `extends` clause, `extends java.lang.Object` (or simply `extends Object`) is added to the class definitions. Furthermore, an implicit `super()` statement is added to the constructors. That is, the class definitions are actually:

```
public class SimpleTime extends Object implements Time {
    public SimpleTime() {
        super(); // Execute the Object constructor
        ...
    }
    ...
}

public class ConsolidatedTime extends Object
implements Time {
    public ConsolidatedTime() {
        super(); // Execute the Object constructor
        ...
    }
    ...
}
```

That is, the classes `SimpleTime` and `ConsolidatedTime` are the subclasses of `Object` and `Time` simultaneously. This is the Java way of implementing multiple inheritances in which a class is the subclass of more than one superclass (a class and an interface). In the Java programming language, a class can only extend one superclass but can implement more than one interface. That is, a class inherits concrete methods from the superclass specified in the `extends` clause and gets the responsibility of implementing the methods imposed by the interface(s) specified in the `implements` clause.

The complete syntax of a class definition is:

```
[public] class class-name extends superclass-name
    [implements interface-name-list] {
        [attribute-declaration]
        [constructor-declaration]
        [method-declaration]
    }
```

Commas separate the interface names if the interface name list contains more than one interface.

## The use of interfaces

Both abstract classes and interface define types with abstract methods. A significant difference is that an interface imposes no restriction on the implementations to the subclasses and provides no implementations to be inherited by the subclasses. Therefore, if the class of an object implements a particular interface, it is guaranteed that the object must define all methods specified by the interface, but the implementation is up to its own implementation.

You can consider an interface as a contract between the calling object and the called object, if the class of the called object implements the interface. Then, the calling object can call any method declared by the interface.

### *Self-test 9.6*

- 1 You are assigned by your boss to write a class for factorial calculation. Besides the documentation that describes the algorithm of calculating the factorial of a number, you are given a Java interface as the following:

```
public interface FactorialCalculator {  
    public long findFactorial(int number);  
}
```

What do you think you are expected to do with the above Java interface? What are the reasons behind the fact that you are given the interface?

- 2 Write classes `IterativeFactorialCalculator`, `CachedFactorialCalculator` and `RecursiveFactorialCalculator` that implement the `FactorialCalculator` interface and obtain the factorials by using a loop, using an array, and recursion respectively.

Discuss the advantages of defining the classes as the subclasses of `FactorialCalculator`.

---

## Appendix F: Window and mouse category events

### Window events

In addition to an Action category event, a Window category event is a common category of event to be handled by GUIs. A typical operation is clicking the <Close> button of a window. Seven event-handler methods corresponding to seven different Window category events are described in Table 9.7.

**Table 9.7** Methods defined by the `WindowListener` interface and the scenarios when the methods are called

Event handler method	The scenario when the event handler method is called
<code>windowOpened()</code>	When the window is first made visible, the method is called.
<code>windowClosed()</code>	A window object acquires some resources from the window system of the machine, and it is possible to call its <code>dispose()</code> method to release those resources. When the <code>dispose()</code> method of a window is called, the <code>windowClosed()</code> method of the registered listener for the Window category is called.
<code>windowClosing()</code>	The method is called if the window is closed by using the operating convention of the window system, such as clicking the close button of the window or pressing Alt-F4 to close a window on a Microsoft windows platform.
<code>windowIconified()</code>	The method is called when the window is minimized, such as the minimize button is clicked.
<code>windowDeiconified()</code>	The method is called when the window is restored after minimizing.
<code>windowActivated()</code>	There is only a single active window on the screen at any time. The active window is changed when you switch among the window applications. When the window becomes the active window, this method is called.
<code>windowDeactivated()</code>	The method is called as soon as the window becomes non-active.

To illustrate the causes of the events and the corresponding event-handler methods, a class `WindowEventDemo` is written in Figure 9.103 to show the sequence of calls to the event-handler methods.

```
// Resolve classes in the java.awt.event package
import java.awt.event.WindowEvent;
import java.awt.event.WindowListener;
import javax.swing.JFrame;

// Definition of class WindowEventDemo
public class WindowEventDemo extends JFrame implements WindowListener {

    // Constructor
    public WindowEventDemo() {
        // Set the frame title
        super("Window Event Demo");

        // Register this WindowEventDemo object as the listener
        // object for Window category events
        addWindowListener(this);

        // Set the default operation when the close button is
        // clicked
        setDefaultCloseOperation(EXIT_ON_CLOSE);
    }

    // Event handler methods
    public void windowActivated(WindowEvent e) {
        System.out.println("windowActivated");
    }

    public void windowClosed(WindowEvent e) {
        System.out.println("windowClosed");
    }

    public void windowClosing(WindowEvent e) {
        System.out.println("windowClosing");
    }

    public void windowDeactivated(WindowEvent e) {
        System.out.println("windowDeactivated");
    }

    public void windowDeiconified(WindowEvent e) {
        System.out.println("windowDeiconified");
    }

    public void windowIconified(WindowEvent e) {
        System.out.println("windowIconified");
    }

    public void windowOpened(WindowEvent e) {
        System.out.println("windowOpened");
    }
}
```

**Figure 9.103** WindowEventDemo.java

A discussion of the `WindowEventDemo` class is provided later in this appendix. A driver program `TestWindowEventDemo` is written that creates a `WindowEventDemo` object and calls its `show()` method. Please find it in the course CD-ROM and website.

Compile the classes and execute the `TestWindowEventDemo` program so you can have a clear understanding which methods of the event-handler object are called — and when.

## Handle a closing window explicitly

If the user is operating the GUI and some data are entered for processing, it is preferable to check whether the entered data are properly saved before terminating the program. In this case, it is necessary to explicitly handle the event when the window is to be closed.

To handle the event explicitly when the window is to be closed, it is necessary to register a `Window` category event-handler object with the window, and the listener object performs suitable operations in the `windowClosing()` method.

To register a `Window` category event-handler object with a window, call the `addWindowListener()` method with supplementary data as the reference to an object whose class is a subclass of the interface `WindowListener`. Similar to handling an `Action` category event, it is possible to create a dedicated object that solely handles the `Window` category events, or the GUI container object bears the responsibility of handling `Window` category events. To write a class so that its object can be a dedicated object for handling `Window` category events, the definition of the class should look like:

```
public class WindowHandler implements WindowListener {
    // Implement all methods defined by the WindowListener interface
    ...
    // Perform the necessary operations, such as validation, and
    // terminate the program if possible
    public void windowClosing(WindowEvent e) {
        ...
    }
    ...
}
```

As the interface `WindowListener` defines seven methods to be implemented by the implementing classes, the implementing classes must define all seven methods even though that `Window` category event is not to be handled (defined as dummy methods). For example:

```
public void windowOpened(WindowEvent e) {}
public void windowClosed(WindowEvent e) {}
public void windowActivated(WindowEvent e) {}
public void windowDeactivated(WindowEvent e) {}
public void windowIconified(WindowEvent e) {}
public void windowDeiconified(WindowEvent e) {}
```

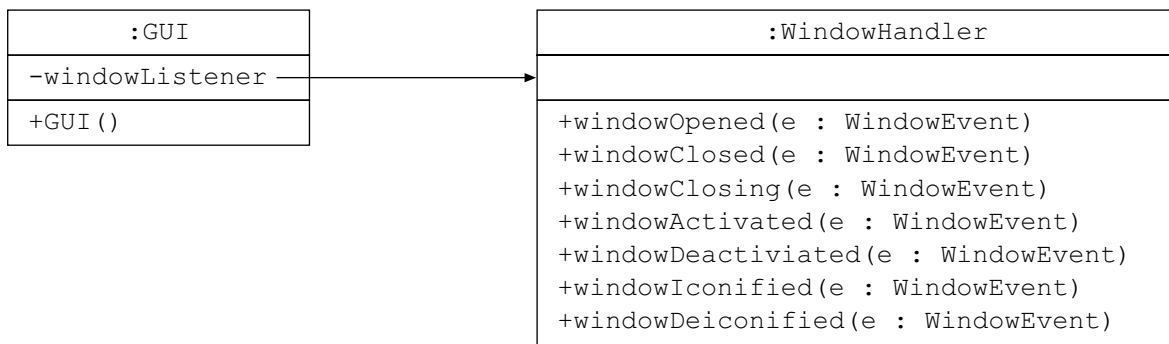
In the constructor of the GUI container, the following statement should be added to register the dedicated object as the Window category event listener:

```
WindowListener listener = new WindowHandler();
addWindowListener(listener);
```

Or, simply:

```
addWindowListener(new WindowHandler());
```

As a result, after registering the listener object, the scenario of the objects is:



**Figure 9.104** The scenario if a dedicated object is registered as the Window category event handler

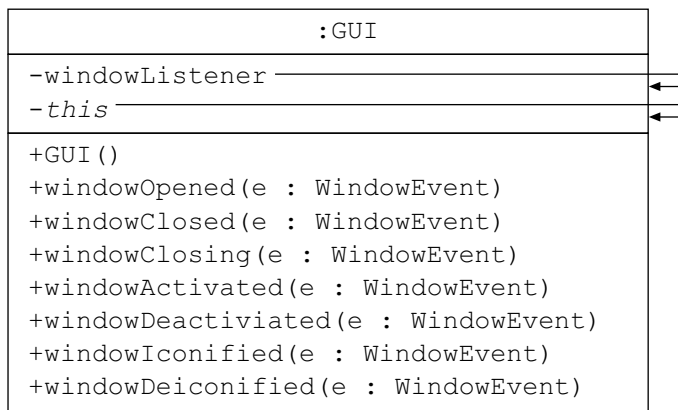
Then, when those corresponding Window category events occur, the dummy event handler methods of the object referred by the variable `windowListener` are called, but no operations are performed.

Another approach is the GUI container acts as the Window category event handler as well. The `WindowEventDemo` class uses such an approach, as shown in Figure 9.103. The class that models the GUI plays the role of a Window category event handler by implementing the interface `WindowListener` and defining all seven methods mandated by the interface. In the constructor, the following statement is required to register the object itself as the Window category event handler:

```
addWindowListener(this);
```

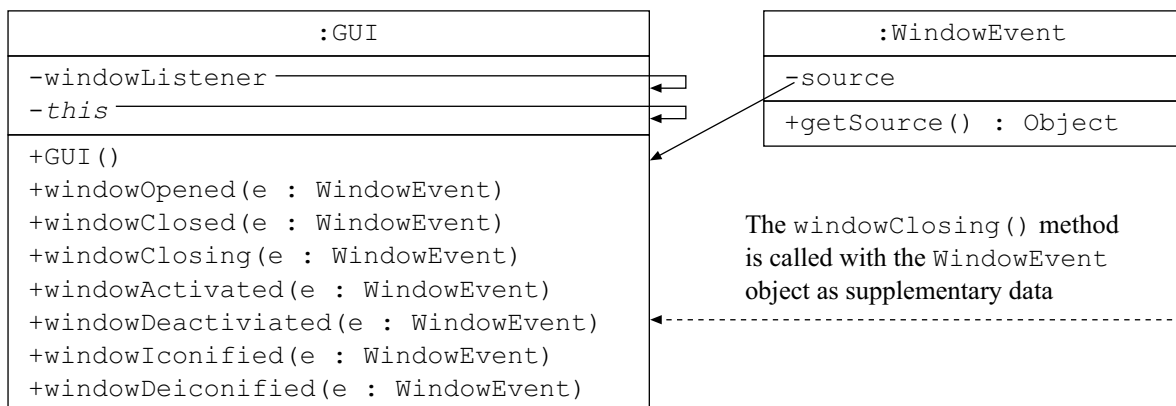
Then, the scenario of the involved object becomes as shown in Figure 9.105.





**Figure 9.105** The scenario of the GUI object if the GUI object itself acts as a listener handler object

Then, whenever a Window category event occurs, a `WindowEvent` object is created and the corresponding event handler method is called. For example, when the <Close> button of the window is clicked, a `WindowEvent` object is created and the `windowClosing()` method of the event-handler object referred by the variable `windowListener` is called, as shown in Figure 9.106.



**Figure 9.106** The `windowClosing()` method is called with a `WindowEvent` object when the close button of the window is clicked

Based on the `SimpleCalculator1` class, a class `SimpleCalculator3` is written that prompts the user before terminating the window, as shown in Figure 9.107.

```

// Resolve classes in java.awt, java.awt.event and javax.swing packages
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

// The definition of the class SimpleCalculator3
public class SimpleCalculator3
    extends JFrame
    implements ActionListener, WindowListener {
    // Attributes
    private JLabel firstLabel = new JLabel("First number: ");
    private JLabel secondLabel = new JLabel("Second number: ");
    private JLabel resultLabel = new JLabel("Result=");
  
```

```
private JTextField firstTextField = new JTextField(15);
private JTextField secondTextField = new JTextField(15);
private JTextField resultTextField = new JTextField(15);
private JButton addButton = new JButton("+");
private JButton subtractButton = new JButton("-");
private JButton multiplyButton = new JButton("*");
private JButton divideButton = new JButton("/");

// Constructor
//
public SimpleCalculator3() {
    // Set the frame title
    super("Simple Calculator");

    // Create the label panel
    JPanel labelPanel = new JPanel();
    labelPanel.setLayout(new GridLayout(3, 1));
    labelPanel.add(firstLabel);
    labelPanel.add(secondLabel);
    labelPanel.add(resultLabel);

    // Create the text field panel
    JPanel textFieldPanel = new JPanel();
    textFieldPanel.setLayout(new GridLayout(3, 1));
    textFieldPanel.add(firstTextField);
    textFieldPanel.add(secondTextField);
    textFieldPanel.add(resultTextField);

    // Make the result text field not to be edited by user
    resultTextField.setEditable(false);

    // Create the button panel
    JPanel buttonPanel = new JPanel();
    buttonPanel.add(addButton);
    buttonPanel.add(subtractButton);
    buttonPanel.add(multiplyButton);
    buttonPanel.add(divideButton);

    // Add all panels to the frame container
    Container contentPane = getContentPane();
    contentPane.add(labelPanel, BorderLayout.WEST);
    contentPane.add(textFieldPanel, BorderLayout.CENTER);
    contentPane.add(buttonPanel, BorderLayout.SOUTH);

    // Register this SimpleCalculator3 object to be the listener
    // for Action category event of the four buttons
    addButton.addActionListener(this);
    subtractButton.addActionListener(this);
    multiplyButton.addActionListener(this);
    divideButton.addActionListener(this);

    // Register this SimpleCalculator3 object to be the listener
    // for the Window category event
    addWindowListener(this);

    // Set the default operation when the close button is
    // clicked
    setDefaultCloseOperation(DO_NOTHING_ON_CLOSE);
```

```

        // Set the GUI to the optimal size
        pack();
    }

    // Called if any button is clicked
    public void actionPerformed(ActionEvent e) {
        // Get the reference of the event source object
        Object source = e.getSource();

        // Get two double values from the first two text fields
        double num1 = Double.parseDouble(firstTextField.getText());
        double num2 = Double.parseDouble(secondTextField.getText());

        if (source == addButton) {
            resultTextField.setText(num1 + num2 + "");
        } else if (source == subtractButton) {
            resultTextField.setText(num1 - num2 + "");
        } else if (source == multiplyButton) {
            resultTextField.setText(num1 * num2 + "");
        } else if (source == divideButton) {
            resultTextField.setText(num1 / num2 + "");
        }
    }

    // Called if the close button of the window is called
    public void windowClosing(WindowEvent e) {
        int result = JOptionPane.showConfirmDialog(
            this,
            "Are you sure to quit?",
            "Simple Calculator",
            JOptionPane.YES_NO_OPTION);
        if (result == JOptionPane.YES_OPTION) {
            System.exit(0);
        }
    }

    // Dummy window event handler methods
    public void windowOpened(WindowEvent e) {}
    public void windowClosed(WindowEvent e) {}
    public void windowIconified(WindowEvent e) {}
    public void windowDeiconified(WindowEvent e) {}
    public void windowActivated(WindowEvent e) {}
    public void windowDeactivated(WindowEvent e) {}
}

```

**Figure 9.107** SimpleCalculator3.java

A change to the class definition is the class SimpleCalculator3 implements two interfaces — ActionListener and WindowListener. The SimpleCalculator3 object is registered with the JButton objects and the window itself to handle the Action category events of the JButton objects and the Window category events of the window object.

```

addButton.addActionListener(this);
subtractButton.addActionListener(this);
multiplyButton.addActionListener(this);
divideButton.addActionListener(this);

addWindowListener(this);

```

You can see that the method `addWindowListener()` is called without specifying any reference variable. The actual interpretation is:

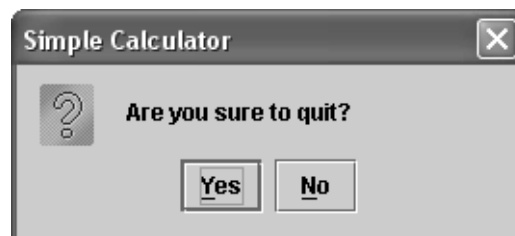
```
this.addWindowListener(this);
```

Therefore, the object itself (supplied as supplementary data to the method by the implicit reference variable `this`) is registered as the Window category event-handler object of the `SimpleCalculator3` object itself (by the implicit reference variable `this`).

Another change is that the default close operation of the `SimpleCalculator3` object is set to `DO_NOTHING_ON_CLOSE`. Then, the window will not close by clicking the close button. It is necessary to close the window explicitly instead.

When the Close button of the window is clicked, the `windowClosing()` method of the object referred to by the `windowListener` variable is called. With respect to a `SimpleCalculator3` object, the object referred by the `windowListener` variable is the `SimpleCalculator3` object itself. Therefore, the `windowClosing()` method of the `SimpleCalculator3` object is called.

Compile the classes `SimpleCalculator3` and `TestSimpleCalculator3` (on the course CD-ROM and website), and execute the `TestSimpleCalculator3` program. A GUI similar to the `SimpleCalculator1` is shown on the screen. Whenever the <Close> button is clicked, the `windowClosing()` method of the `SimpleCalculator3` object is called and the dialog in Figure 9.108 is shown on the screen to confirm termination of the program.



**Figure 9.108** A confirmation dialog is shown to confirm termination of the program

If the user clicks the <Yes> button on the confirmation dialog, the method `showConfirmDialog()` returns a value of `YES_OPTION`. Then, the `exit()` method is called to terminate the program. Otherwise, no operation is carried out and the program execution continues.

Please use the following self-test to experiment with handling window events.

## Self-test 9.7

Modify the class `RegionSelectionFrame` (Figure 9.52) so that it shows the user selection when the window is to be closed.

## Mouse events

Mouse movements are events to the GUI, such as clicking the mouse button while the mouse pointer is located within the boundary of a GUI component, the mouse pointer entering or leaving the boundary of a GUI component, and moving the mouse pointer on a GUI component. These operations are handled as events to the GUI components and are categorized into *mouse category* and *mouse motion category* respectively. Please refer to Table 9.2 for the events and the corresponding methods to be called for these two event categories.

Similar to handling the action and window categories, it is necessary to create an object of a class that implements `MouseListener` and/or `MouseMotionListener`. Then, call the `addMouseListener()` and `addMouseMotionListener()` with the reference of the object as the supplementary data to register the object as the listener for mouse category events and mouse motion category events respectively.

A sample class `GUIWithEvent4`, which can handle mouse category and mouse motion category events, is written in Figure 9.109.

```
// Resolve classes in java.awt.event and javax.swing packages
import java.awt.event.*;
import javax.swing.*;

// Definition of class GUIWithEvent4
public class GUIWithEvent4 extends JFrame
    implements MouseListener, MouseMotionListener {

    public GUIWithEvent4() {
        // Set the frame title
        super("GUI with Event");

        // Register itself as the listener for mouse and mouse
        // motion listener
        addMouseListener(this);
        addMouseMotionListener(this);

        // Set the default operation if the close button of the frame
        // is clicked
        setDefaultCloseOperation(EXIT_ON_CLOSE);

        // Set the size to be 100 by 100 explicitly
        setSize(100, 100);
    }
}
```

```

// Method called if the mouse button is clicked within the frame
public void mouseClicked(MouseEvent e) {
    System.out.println("Mouse button clicked at (" +
        e.getX() + "," + e.getY() + ")");
}

// Method called if the mouse button is pressed within the frame
public void mousePressed(MouseEvent e) {
    System.out.println("Mouse button pressed at (" +
        e.getX() + "," + e.getY() + ")");
}

// Method called if the mouse button is released within the frame
public void mouseReleased(MouseEvent e) {
    System.out.println("Mouse button released at (" +
        e.getX() + "," + e.getY() + ")");
}

// Method called if the mouse pointer entered the boundary
// of the frame
public void mouseEntered(MouseEvent e) {
    System.out.println("Mouse pointer entered the frame");
}

// Method called if the mouse pointer left the boundary
// of the frame
public void mouseExited(MouseEvent e) {
    System.out.println("Mouse pointer left the frame");
}

// Method called if the mouse pointer is dragged within
// the boundary of the frame
public void mouseDragged(MouseEvent e) {
    System.out.println("Mouse button dragged at (" +
        e.getX() + "," + e.getY() + ")");
}

// Method called if the mouse pointer is moved within
// the boundary of the frame
public void mouseMoved(MouseEvent e) {
    System.out.println("Mouse button moved at (" +
        e.getX() + "," + e.getY() + ")");
}
}

```

**Figure 9.109** GUIWithEvent4.java

It is possible to register mouse and mouse motion category event listeners with a GUI container and a GUI component. For example, a GUIWithEvent4 object, which is also a JFrame object, registers itself to be its mouse and mouse motion listeners.

A driver program, TestGUIWithEvent4, is written to test the GUIWithEvent4 class. Please refer to the course CD-ROM and website for the class definition. Compile the classes and execute the

TestGUIWithEvent4 program. A frame as shown in Figure 9.110 will appear on the screen.



**Figure 9.110** The frame shown by executing the TestGUIWithEvent4 program

The GUIWithEvent4 object will handle all mouse-related operations. Corresponding messages are shown on the screen, such as:

```
Mouse pointer entered the frame
Mouse button moved at (118,77)
Mouse button moved at (117,77)
Mouse button moved at (114,76)
Mouse button moved at (113,76)
Mouse button moved at (114,76)
Mouse button moved at (116,77)
Mouse button moved at (118,78)
Mouse pointer left the frame
```

A TestGUIWithEvent4 object only shows information about the mouse category event that occurred. A practical use of handling mouse and mouse motion category events is presented in the drawing software in Appendix H.

## Appendix G: A simple calculator

In this appendix we are going to write classes that model a simple calculator, to illustrate how to handle events that are caused by different GUI components on a GUI. The calculator to be written is as shown in Figure 9.111.

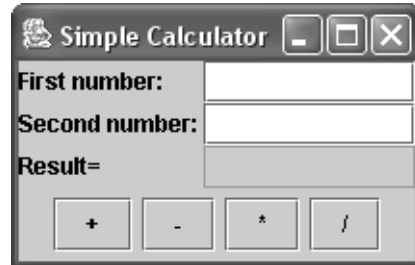


Figure 9.111 A simple calculator

The calculator shown in Figure 9.111 contains labels, text fields and buttons. All labels and all text fields are added to panels with a `GridLayout` layout manager that has three rows and one column. All buttons are added to a panel with a `FlowLayout` layout manager. Finally, the panels for labels, text fields, and buttons are added to the west, center, and south regions of the content pane respectively.

To handle the event when a button is clicked, a possible way is to equip the class that models the calculator with the ability to handle the `Action` category event. That is, the class has to implement the `ActionListener` interface and define the `actionPerformed()` method.

The complete definition of the class, `SimpleCalculator1`, is written as shown in Figure 9.112.

```
// Resolve classes in java.awt, java.awt.event and javax.swing packages
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

// The definition of the class SimpleCalculator1
public class SimpleCalculator1
    extends JFrame
    implements ActionListener {
    // GUI components
    // Labels
    private JLabel firstLabel = new JLabel("First number: ");
    private JLabel secondLabel = new JLabel("Second number: ");
    private JLabel resultLabel = new JLabel("Result=");
    // Text fields
    private JTextField firstTextField = new JTextField(10);
    private JTextField secondTextField = new JTextField(10);
    private JTextField resultTextField = new JTextField(10);
    // Buttons
    private JButton addButton = new JButton("+");
    private JButton subtractButton = new JButton("-");
```



```
private JButton multiplyButton = new JButton("*");
private JButton divideButton = new JButton("/");

// Constructor
//
public SimpleCalculator1() {
    // Set the frame title
    super("Simple Calculator");

    // Create the label panel
    JPanel labelPanel = new JPanel();
    labelPanel.setLayout(new GridLayout(3, 1));
    labelPanel.add(firstLabel);
    labelPanel.add(secondLabel);
    labelPanel.add(resultLabel);

    // Create the text field panel
    JPanel textFieldPanel = new JPanel();
    textFieldPanel.setLayout(new GridLayout(3, 1));
    textFieldPanel.add(firstTextField);
    textFieldPanel.add(secondTextField);
    textFieldPanel.add(resultTextField);
    // Make the result text field not to be edited by user
    resultTextField.setEditable(false);

    // Create the button panel
    JPanel buttonPanel = new JPanel();
    buttonPanel.add(addButton);
    buttonPanel.add(subtractButton);
    buttonPanel.add(multiplyButton);
    buttonPanel.add(divideButton);

    // Add all panels to the frame container
    Container contentPane = getContentPane();
    contentPane.add(labelPanel, BorderLayout.WEST);
    contentPane.add(textFieldPanel, BorderLayout.CENTER);
    contentPane.add(buttonPanel, BorderLayout.SOUTH);

    // Register this SimpleCalculator1 object to be the listener
    // for Action category event of the four buttons
    addButton.addActionListener(this);
    subtractButton.addActionListener(this);
    multiplyButton.addActionListener(this);
    divideButton.addActionListener(this);

    // Set the default operation when the close button is
    // clicked
    setDefaultCloseOperation(EXIT_ON_CLOSE);

    // Set the GUI to the optimal size
    pack();
}

// Method called if a button is clicked
public void actionPerformed(ActionEvent e) {
    // Get the reference of the event source object
    Object source = e.getSource();
```

```

        // Get two double values from the first two text fields
        double num1 = Double.parseDouble(firstTextField.getText());
        double num2 = Double.parseDouble(secondTextField.getText());

        if (source == addButton) {
            resultTextField.setText(num1 + num2 + "");
        } else if (source == subtractButton) {
            resultTextField.setText(num1 - num2 + "");
        } else if (source == multiplyButton) {
            resultTextField.setText(num1 * num2 + "");
        } else if (source == divideButton) {
            resultTextField.setText(num1 / num2 + "");
        }
    }
}

```

**Figure 9.112** SimpleCalculator1.java

The steps in arranging the GUI components should be obvious to you now. After the GUI components are added to the panel and subsequently to the content pane, each button is registered with the SimpleCalculator1 object as its Action category event listener. Therefore, if any button gets clicked, the `actionPerformed()` method of the SimpleCalculator1 object is called.

The design of the `actionPerformed()` method is that it obtains the event source from the event object first by calling the `getSource()` method of the supplied `ActionEvent` object. By comparing the reference of the event source with the references of the four `JButton`, it is possible to determine the button that was clicked. Finally, a nested `if/else` statement is used to perform the calculation and set the result text field accordingly.

A driver program, `TestSimpleCalculator1` class, is written and is available from the course CD-ROM and website. Compile the classes and execute the `TestSimpleCalculator1` program. A GUI-based calculator as shown in Figure 9.111 appears on the screen.

The `actionPerformed()` method for the SimpleCalculator1 class is relatively trivial and simple. However, if there are a lot of GUI components to be handled by the event-handler method and the operations for each event source are complicated, the `if/else` statement in the `actionPerformed()` method will become complicated and unmanageable.

An enhanced version of the SimpleCalculator1 class, the SimpleCalculator2 class that uses a better approach to handle events caused by different event sources, is shown in Figure 9.113.

```
// Resolve classes in java.awt, java.awt.event and javax.swing packages
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

// The definition of the class SimpleCalculator2
public class SimpleCalculator2
    extends JFrame
    implements ActionListener {
    // GUI components
    // Labels
    private JLabel firstLabel = new JLabel("First number: ");
    private JLabel secondLabel = new JLabel("Second number: ");
    private JLabel resultLabel = new JLabel("Result=");
    // Text fields
    private JTextField firstTextField = new JTextField(10);
    private JTextField secondTextField = new JTextField(10);
    private JTextField resultTextField = new JTextField(10);
    // Buttons
    private JButton addButton = new JButton("+");
    private JButton subtractButton = new JButton("-");
    private JButton multiplyButton = new JButton("*");
    private JButton divideButton = new JButton("/");
    // Constructor
    //
    public SimpleCalculator2() {
        // Set the frame title
        super("Simple Calculator");

        // Create the label panel
        JPanel labelPanel = new JPanel();
        labelPanel.setLayout(new GridLayout(3, 1));
        labelPanel.add(firstLabel);
        labelPanel.add(secondLabel);
        labelPanel.add(resultLabel);

        // Create the text field panel
        JPanel textFieldPanel = new JPanel();
        textFieldPanel.setLayout(new GridLayout(3, 1));
        textFieldPanel.add(firstTextField);
        textFieldPanel.add(secondTextField);
        textFieldPanel.add(resultTextField);
        // Make the result text field not to be edited by user
        resultTextField.setEditable(false);

        // Create the button panel
        JPanel buttonPanel = new JPanel();
        buttonPanel.add(addButton);
        buttonPanel.add(subtractButton);
        buttonPanel.add(multiplyButton);
        buttonPanel.add(divideButton);
```

```

        // Add all panels to the frame container
        Container contentPane = getContentPane();
        contentPane.add(labelPanel, BorderLayout.WEST);
        contentPane.add(textFieldPanel, BorderLayout.CENTER);
        contentPane.add(buttonPanel, BorderLayout.SOUTH);

        // Register this SimpleCalculator1 object to be the listener
        // for Action category event of the four buttons
        addButton.addActionListener(this);
        subtractButton.addActionListener(this);
        multiplyButton.addActionListener(this);
        divideButton.addActionListener(this);

        // Set the default operation when the close button is
        // clicked
        setDefaultCloseOperation(EXIT_ON_CLOSE);

        // Set the GUI to the optimal size
        pack();
    }

    public void actionPerformed(ActionEvent e) {
        // Get the reference of the event source object
        Object source = e.getSource();

        if (source == addButton) {
            addButtonClicked(e);
        } else if (source == subtractButton) {
            subtractButtonClicked(e);
        } else if (source == multiplyButton) {
            multiplyButtonClicked(e);
        } else if (source == divideButton) {
            divideButtonClicked(e);
        }
    }

    // The operation to be carried out when the add button is clicked
    private void addButtonClicked(ActionEvent e) {
        double num1 = Double.parseDouble(firstTextField.getText());
        double num2 = Double.parseDouble(secondTextField.getText());
        resultTextField.setText(num1 + num2 + "");
    }

    // The operation to be carried out when the subtract button
    // is clicked
    private void subtractButtonClicked(ActionEvent e) {
        double num1 = Double.parseDouble(firstTextField.getText());
        double num2 = Double.parseDouble(secondTextField.getText());
        resultTextField.setText(num1 - num2 + "");
    }

```

```

// The operation to be carried out when the multiply button
// is clicked
private void multiplyButtonClicked(ActionEvent e) {
    double num1 = Double.parseDouble(firstTextField.getText());
    double num2 = Double.parseDouble(secondTextField.getText());
    resultTextField.setText(num1 * num2 + "");
}

// The operation to be carried out when the divide button
// is clicked
private void divideButtonClicked(ActionEvent e) {
    double num1 = Double.parseDouble(firstTextField.getText());
    double num2 = Double.parseDouble(secondTextField.getText());
    resultTextField.setText(num1 / num2 + "");
}
}

```

**Figure 9.113** SimpleCalculator2.java

Compared with the SimpleCalculator1 class, although the SimpleCalculator2 class is longer, the operations to be performed when a button is clicked are encapsulated in individual methods. Each method accepts supplementary data as the reference of the ActionEvent object so that the method can obtain the data about the event, if necessary. Furthermore, the methods are marked `private` so that they are to be called by the methods of the same class only.

As the approach used by the SimpleCalculator2 makes the class definition longer, it clearly isolates the operations to be performed for different event sources. The class definition is therefore more easily maintained. By consolidating the discussions on constructing GUIs and event handling, a recommended template for creating a GUI is shown in Figure 9.114.

```

// import statements for class resolutions
import ...;

public class ClassName extends JFrame implements EventListenerList {
    // An attribute for each GUI components to be added to the
    // content pane
    ...
    // Attributes for other helper objects, such as dialog
    ...
    // Attributes for the data to be processed

    public ClassName() {
        // Set the frame title
        super(...);

        // Add the GUI components to the content pane
        ...

        // Register event handler (this) with the GUI components
        ...
    }
}

```

```
        |
        // Set the default operation when the Close button of the
        // frame is clicked
        setDefaultCloseOperation(...);

        // Set the frame to its optimal size
        pack();
    }

    // Event handler methods
    public void eventOccurred(EventClass e) {
        // If necessary, call the corresponding method for a event
        // source
        ...
    }

    ...

    // Method for each event source
    private void eventSourceAccessed(EventClass e) {
        // Perform the operations for an event source
        ...
    }

    ...

    // Other helper or utility methods
    ...
}
```

**Figure 9.114** A template of the recommended approach to write a class for a GUI

## Appendix H: A drawing program

In this appendix, we discuss how to write a drawing program that enables the user to draw a picture within a frame with the mouse. When the user drags the mouse within a frame, the path of the mouse movement is drawn.

First of all, let's investigate the events caused by the user interactions with the GUI.

- 1 When the user moves the mouse pointer so that it enters the boundary of the GUI, a mouse-entered event (a mouse category event) occurs, but it is not necessary to handle it.
- 2 When the user keeps on moving the mouse pointer within the GUI, mouse-moved events (mouse motion category events) occur but are not handled.
- 3 When the user presses the mouse button to start drawing, a mouse-pressed event (a mouse category event) occurs that becomes the starting point of the path to be drawn. It is then necessary to keep the coordinates when (where) the mouse point is clicked as a reference point by calling the `getX()` and `getY()` methods of the created `MouseEvent` object.
- 4 Then, the user keeps on moving the mouse pointer while the mouse button is pressed. Mouse-dragged events (mouse motion category events) occur, and it is necessary to draw a line from the reference point to the current position to which the mouse is dragged. After the line is drawn, the current position is now treated as the reference point.
- 5 When the user releases the mouse button, a mouse-released event (a mouse category event) occurs. That is the end of the path to be drawn and the reference point is reset. The state is now returned to step 2 in this list.
- 6 When the user moves the mouse pointer out of the GUI, a mouse-exited event (a mouse category event) occurs, and it need not be handled.

Based on the above design, a class `DrawingFrame1` is shown in Figure 9.115.

```
// Resolve classes in java.awt, java.awt.event and javax.swing packages
import java.awt.*;
import java.awt.event.*;

import javax.swing.JFrame;

// Definition of class DrawingFrame1
public class DrawingFrame1
    extends JFrame
    implements MouseListener, MouseMotionListener {
    // The value indicates the start of a sequence of points
```

```

private static final int START_POINT = -1;
// The coordinates of the line to be drawn
private int refXPos = START_POINT;
private int refYPos = START_POINT;
private int xPos, yPos;

// Constructor
public DrawingFrame1() {
    // Set the frame title
    super("Drawing Frame");

    // Add the object itself as the MouseMotionListener
    addMouseMotionListener(this);
    addMouseListener(this);

    // Set the default operation when the close button of the frame
    // is clicked
    setDefaultCloseOperation(EXIT_ON_CLOSE);

    // Set the frame size explicitly
    setSize(200, 200);
}

// The method to be called when the repaint() method is called or
// the frame is exposed
public void paint(Graphics g) {
    // If it is not the start of the path to be drawn, draw a
    // line from the reference point to the current position
    if (refXPos != START_POINT && refYPos != START_POINT) {
        g.drawLine(refXPos, refYPos, xPos, yPos);
    }
    // Update the reference point
    refXPos = xPos;
    refYPos = yPos;
}

// Method called when the mouse button is clicked.
// Set the reference point
public void mousePressed(MouseEvent e) {
    refXPos = e.getX();
    refYPos = e.getY();
}

// Method called when the mouse button is released.
// Reset the reference point
public void mouseReleased(MouseEvent e) {
    refXPos = START_POINT;
    refYPos = START_POINT;
}

// Method called if the mouse is dragged within the frame
public void mouseDragged(MouseEvent e) {
    // Set the point to be drawn
    xPos = e.getX();
    yPos = e.getY();
    // Force the GUI to refresh by calling the paint method
    repaint();
}

// The following are the methods mandated by the interfaces but
// the events are not handled.

```



```

// Method mandated by MouseMotionListener
public void mouseMoved(MouseEvent e) {}
// Method mandated by MouseListener
public void mouseClicked(MouseEvent e) {}
public void mouseEntered(MouseEvent e) {}
public void mouseExited(MouseEvent e) {}
}

```

**Figure 9.115** DrawingFrame1.java

The `mousePressed()` and `mouseReleased()` is called when the mouse button is pressed and released respectively. When the mouse button is pressed, the reference point is updated to the current mouse position kept by the `MouseEvent` object. When the mouse button is released, however, the reference point is reset.

When the mouse is dragged, the `mouseDragged()` method is called, and the current coordinates maintained by the `MouseEvent` object are assigned to the attributes `xPos` and `yPos` respectively. Then, the `repaint()` method is called. Whenever the GUI component is exposed on the screen and the `repaint()` method is called, the `paint()` method of the GUI component is indirectly called. As a result, the `paint()` method draws a line on the screen using the `Graphics` object supplied to the method, and the current coordinates are kept as the reference point.

A driver program `TestDrawingFrame1` is written. It is available on the course CD-ROM and website. Compile the classes and execute the `TestDrawingFrame1`, and you can test the `DrawingFrame1` object. Figure 9.116 is a sample execution of the program.



**Figure 9.116** A sample execution of the `TestDrawingFrame1`

You are now happy with the `DrawingFrame1` class and can draw whatever you like on the GUI. However, sooner or later you will find that the program exhibits a deficiency. Whenever the GUI is minimized and restored, the drawing on the GUI is erased. The reason is that the `DrawingFrame1` only maintains the coordinates of the last line to be drawn. Therefore, when you restore the GUI, only the last line is drawn.

To tackle the program, it is necessary to keep the coordinates of all lines to be drawn. An enhanced version of the drawing program is presented in the following subsection.

### An enhanced drawing program

To keep the coordinates of a line, two points (and hence four values of type `int`) have to be maintained. A new class, the `Line` class, is written to maintain the coordinates of the lines.

```
// Definition of class Line
public class Line {
    // The attributes for coordinates
    private int startX;
    private int startY;
    private int endX;
    private int endY;

    // Constructor
    public Line(int startX, int startY, int endX, int endY) {
        this.startX = startX;
        this.startY = startY;
        this.endX = endX;
        this.endY = endY;
    }

    // Getter methods for the coordinates

    public int getEndX() {
        return endX;
    }

    public int getEndY() {
        return endY;
    }

    public int getStartX() {
        return startX;
    }

    public int getStartY() {
        return startY;
    }
}
```

**Figure 9.117** `Line.java`

Each `Line` object maintains the coordinates of a line. As it is not necessary to update the coordinates maintained by a `Line` object after the object is created, the `Line` class is written to be immutable and only getter methods are defined.

With the `Line` class, an enhanced version of the drawing program, `DrawingFrame2` is shown in Figure 9.118.

```
// Resolve classes in java.awt, java.awt.event and javax.swing packages
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

// Definition of class DrawingFrame2
public class DrawingFrame2
    extends JFrame
    implements MouseListener, MouseMotionListener {
    private static final int START_POINT = -1;
    // The points to be drawn
    private Line[] lines = new Line[100];
    // The number of points to be drawn
    private int lineCount = 0;
    private int refXPos = START_POINT;
    private int refYPos = START_POINT;
    private int xPos, yPos;

    // Constructor
    public DrawingFrame2() {
        // Set the frame title
        super("Drawing Frame");

        // Add the object itself as the MouseMotionListener
        addMouseMotionListener(this);
        addMouseListener(this);

        // Set the default operation when the close button of the
        // frame is clicked
        setDefaultCloseOperation(EXIT_ON_CLOSE);

        // Set the frame size explicitly
        setSize(200, 200);
    }

    // The method to be called when the repaint() method is called or
    // the frame is exposed
    public void paint(Graphics g) {
        // Draw the lines one by one
        for (int i = 0; i < lineCount; i++) {
            Line line = lines[i];
            g.drawLine(
                line.getStartX(),
                line.getStartY(),
                line.getEndX(),
                line.getEndY());
        }
    }

    // Method called when the mouse button is clicked.
    // Set the reference point
    public void mousePressed(MouseEvent e) {
        refXPos = e.getX();
        refYPos = e.getY();
    }
}
```

```

// Method called when the mouse button is released.
// Reset the reference point
public void mouseReleased(MouseEvent e) {
    refXPos = START_POINT;
    refYPos = START_POINT;
}

// Method called if the mouse is dragged within the frame
public void mouseDragged(MouseEvent e) {
    // Set the point to be drawn
    xPos = e.getX();
    yPos = e.getY();

    // Add a new Line object if it is not a starting point
    if (refXPos != START_POINT && refYPos != START_POINT) {
        // If all array elements are used, create a new array
        // object with double size of the current one
        if (lineCount >= lines.length) {
            Line[] temp = new Line[lines.length * 2];
            System.arraycopy(lines, 0, temp, 0, lines.length);
            lines = temp;
        }
        // Add the new Line to the array object
        lines[lineCount] = new Line(refXPos, refYPos, xPos, yPos);
        // Increase the number of Line maintained
        lineCount++;
    }

    // Update the reference point
    refXPos = xPos;
    refYPos = yPos;
    // Force the GUI to refresh itself and the paint() method
    // will be executed
    repaint();
}

// The following are the methods mandated by the interfaces but
// the events are not handled.

// Method mandated by MouseMotionListener
public void mouseMoved(MouseEvent e) {}
// Method mandated by MouseListener
public void mouseClicked(MouseEvent e) {}
public void mouseEntered(MouseEvent e) {}
public void mouseExited(MouseEvent e) {}
}

```

**Figure 9.118** DrawingFrame2.java

The `mouseDragged()` method of the class `DrawingFrame2` is modified to add a `Line` object to the array object, and the `paint()` method is enhanced to draw the lines maintained by the array object. Therefore, every time the `paint()` method is called, all lines maintained by the array object are redrawn. The drawing survives when the GUI is minimized and then restored.

A driver program, `TestDrawingFrame2` class, is written and is available from the course CD-ROM and website. Compile the classes and execute the `TestDrawingFrame2`. A GUI similar to `DrawingFrame1` appears on the screen. You can now test it. You will find that the drawing can be restored after the GUI is re-exposed.

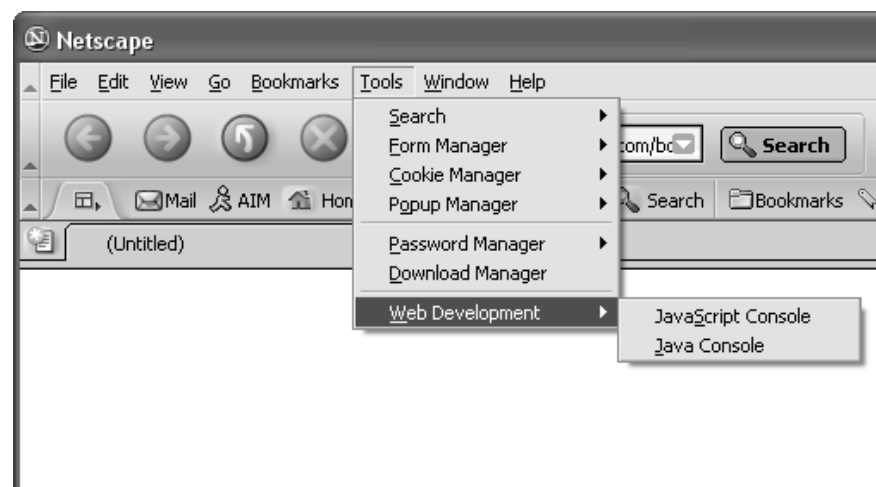
## Appendix I: Showing Java consoles of Web browsers

Java applets are compiled Java class files to be executed by the JVM of a Java-enabled Web browser. While a Java applet is executing, it is possible to show messages via the standard output with the method `System.out.println()`. If a Java applet is launched by executing the `appletviewer` software, the messages are shown in the Command Prompt. In common Web browsers — Netscape and Microsoft Internet Explorer — the messages are shown in the Java console but are not shown by default.

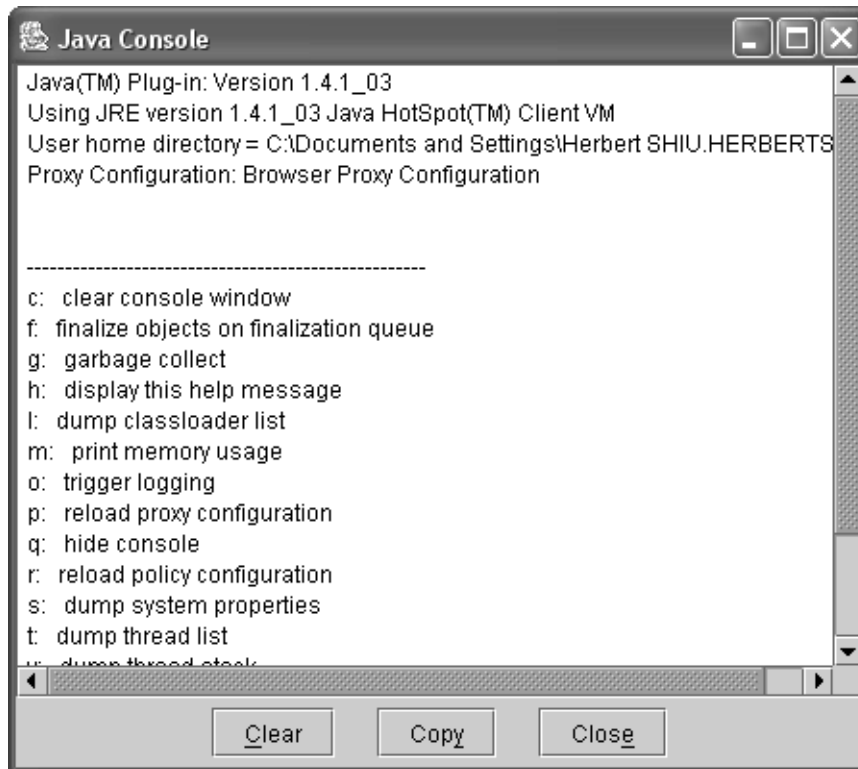
The following subsections provide you with the procedures to show the Java console with Netscape and Microsoft Internet Explorer.

### Netscape

Start the Netscape browser and select <Tools>, <Web Development> and then <Java Console> as shown in Figure 9.119. The Java Console as shown in Figure 9.120 will be shown on the screen.



**Figure 9.119** Show the Java Console with the Netscape browser



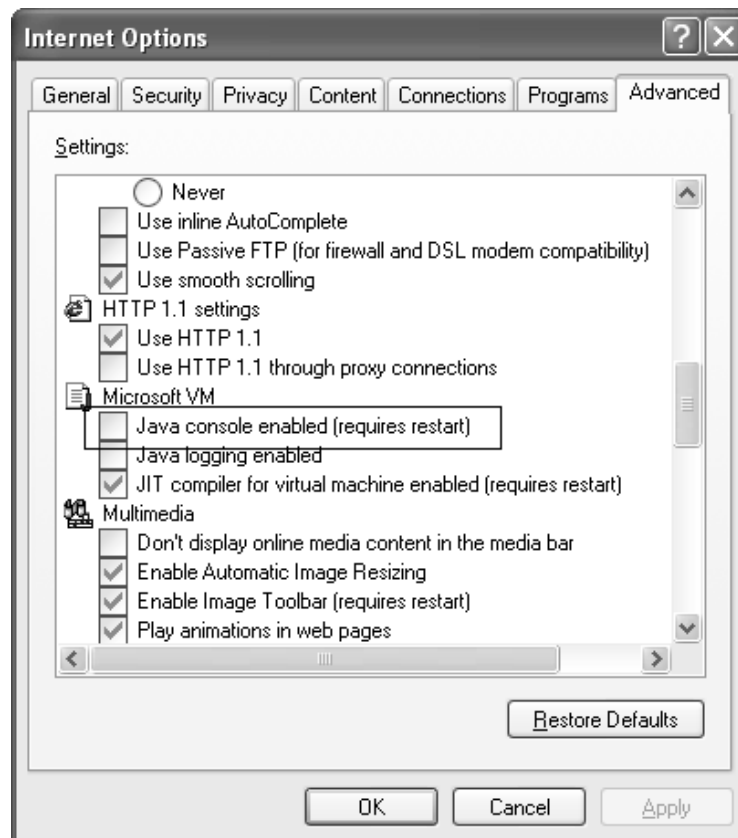
**Figure 9.120** Show the Java Console with the Netscape browser

The Java Console enables you to read the messages shown by the standard output. Also, it provides a menu that you can use to control the operation of the JVM.

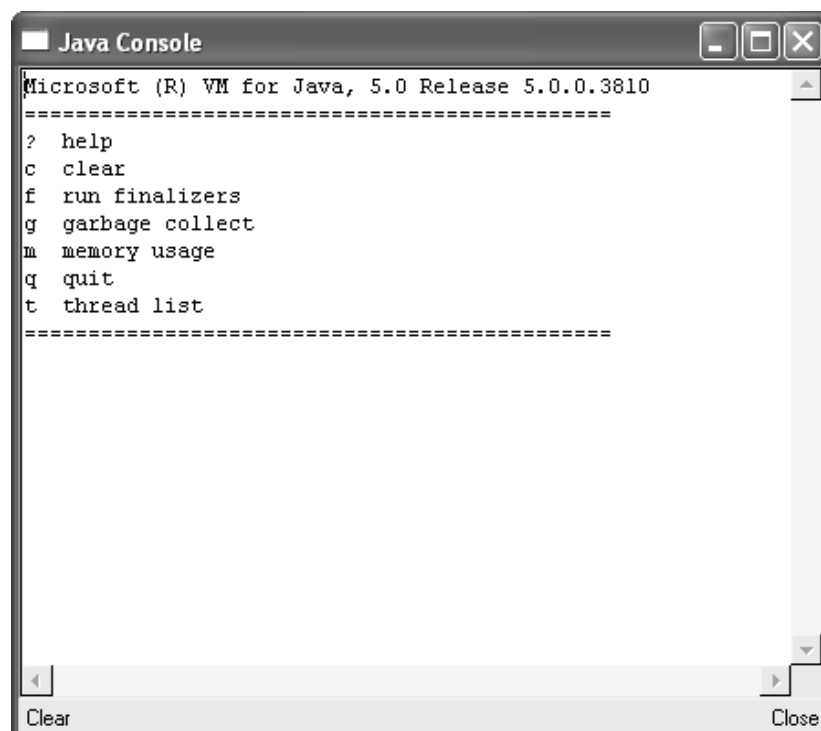
## Microsoft Internet Explorer

Microsoft Internet Explorer supports two types of JVM provided by Microsoft and Sun Microsystems. If you have installed the Java SDK, the Sun JVM will be added to Internet Explorer. Then, you start the Java Console by selecting Sun Java Console from the <Tools> menu of Internet Explorer. You can further verify whether a JVM by Microsoft is installed in the Internet Explorer. The steps to show the Java Console of the Microsoft version of JVM are:

- 1 Select the <Internet Options> from the <Tools> menu of the Internet Explorer to show the <Internet Options> dialog.
- 2 Click the <Advanced> tag.
- 3 Scroll the options list to find and select the option <Java console enabled> (requires restart).



- 4 Click the <OK> button.
- 5 Restart Internet Explorer.
- 6 Now, you can select the option <Java Console> from the <View> menu of Internet Explorer.





## Appendix J: Writing a GUI-based payroll software application

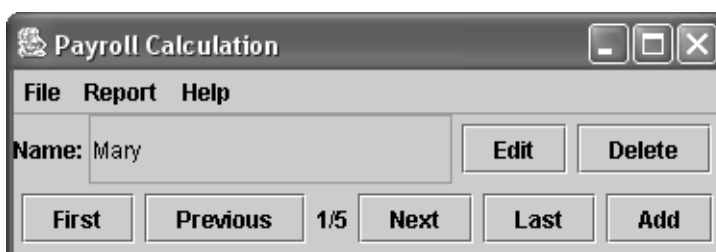
We have been using the payroll software application since *Unit 7* to illustrate a real-world application of the knowledge you acquired in that unit. In *Unit 7*, you learned object-oriented programming in the Java programming language. Objects of classes that formed a class hierarchy (with respect to inheritance) modelled the staff members in a company. You learned how to perform input/output operations with the Java programming language so that staff payroll data can be maintained on a computer drive. Furthermore, with the knowledge of using the standard input, output and error streams, an interactive text-mode payroll software was presented in *Unit 7*.

In this unit, you learned how to build GUIs with the Java programming language and the Swing packages that are provided by the standard software library. Such knowledge enables us to write a GUI-based payroll software application that is even more user-friendly. In this appendix, a GUI-based payroll software application is introduced so that you can see how to design a practical GUI-based software application. Furthermore, some classes written in previous units are used. You may appreciate the beauty of the reusability enabled by object-oriented programming.

### Introduction to the GUI-based payroll software application

Before we start discussing how to write the payroll software, it's preferable to have an idea of how the payroll software operates and the sequence of operations so that you can figure out which GUI objects are required.

Execute the payroll software application. The main frame of the application is shown on the screen. A possible one is shown in Figure 9.121.



**Figure 9.121** The main frame of the payroll software

The software maintains a list of `Staff` objects during execution. A main field shows the name of the current staff. By using the <First>, <Previous>, <Next> and <Last> buttons, the user can navigate the list of `Staff` objects. Between the <Previous> and <Next> buttons, there is a

label indicating the `Staff` record to be handled. The format is `<Current staff number>/<Total staff number>`.

`<Edit>` and `<Delete>` enables the user to edit the current `Staff` object or remove the current `Staff` object from the list. To add and append a new `Staff` object to the list, click the `<Add>` button.

The File menu enables the user to read and write staff data file, and the Report menu provides options to show the MPF and payroll report. The `<Help>` menu only shows some information about the program.

To add a new `Staff` object by clicking the `<Add>` button, a dialog is shown on the screen for the user to select the staff type, as shown in Figure 9.122.



Figure 9.122 A staff type selection dialog

In the staff type selection dialog shown in Figure 9.122, radio boxes are used to determine the staff type required. Once a staff type is selected and the `<OK>` button is clicked, a suitable dialog is shown on the screen for the user to enter the necessary data for the `Staff` object. For example, three different dialogs as shown in Figure 9.123 to Figure 9.125 are used to obtain the Staff data for clerk, salesperson, and manager.



Figure 9.123 The dialog for getting clerk data

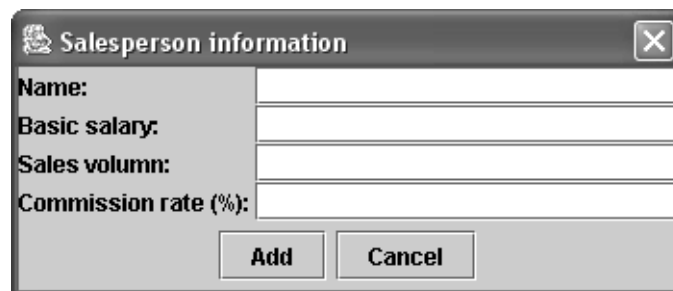


Figure 9.124 The dialog for getting salesperson data

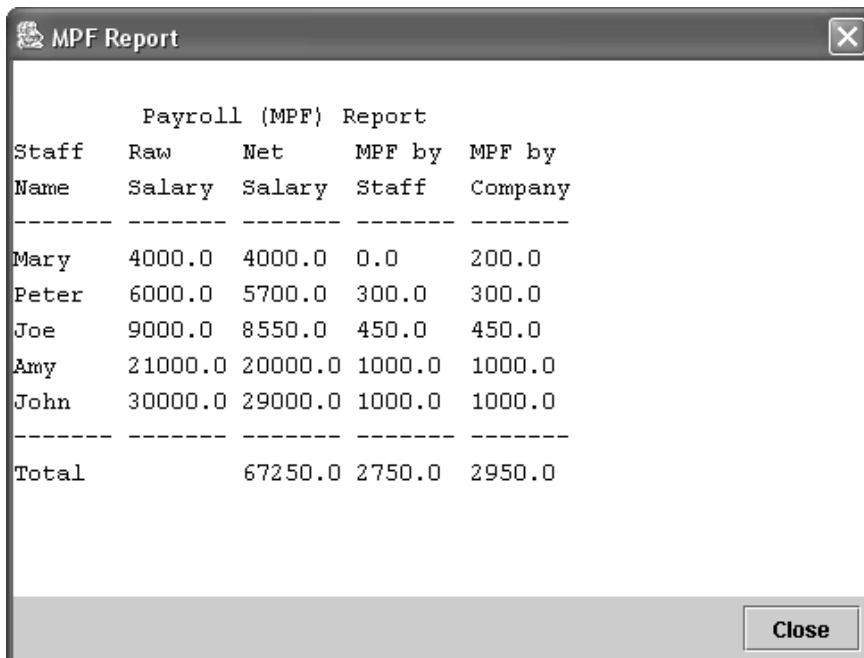


A dialog box titled "Manager information" with a close button (X) in the top right corner. It contains three input fields labeled "Name:", "Basic salary:", and "Allowance:". Below the input fields are two buttons: "Add" and "Cancel".

**Figure 9.125** The dialog for getting manager data

Furthermore, the dialogs shown in Figure 9.123 to Figure 9.125 can be used to edit existing `Staff` objects.

As GUI-based software, the MPF and payroll report is to be shown in a frame or dialog. For example, a report dialog as shown in Figure 9.126 is used for showing the report.



A dialog box titled "MPF Report" with a close button (X) in the top right corner. It displays a payroll report table. The table has five columns: Staff Name, Raw Salary, Net Salary, MPF by Staff, and MPF by Company. The data rows are for Mary, Peter, Joe, Amy, and John, followed by a Total row. A "Close" button is located at the bottom right of the dialog.

Payroll (MPF) Report				
Staff Name	Raw Salary	Net Salary	MPF by Staff	MPF by Company
Mary	4000.0	4000.0	0.0	200.0
Peter	6000.0	5700.0	300.0	300.0
Joe	9000.0	8550.0	450.0	450.0
Amy	21000.0	20000.0	1000.0	1000.0
John	30000.0	29000.0	1000.0	1000.0
Total		67250.0	2750.0	2950.0

**Figure 9.126** An MPF and payroll report dialog

The resultant program produces the screen shots of the frame and dialogs shown above. If you design a GUI-based application from scratch, you may draft the layouts of the frames/dialogs on a piece of paper and then consider how to implement them.

## The implementation of the software

Based on the design mentioned in the previous subsection, we can now consider how to implement them one by one. In this subsection, the frames or dialogs are discussed from the simpler to the more complex ones.

## The report dialog

Figure 9.126 is the simplest dialog of all. It contains two GUI components only, a read-only text area and a close button. It can be built with a `BorderLayout` layout manager object so that the text area occupies the center region and the button occupies the south region. However, such an arrangement makes the `<Close>` button occupy the entire south region and hence become a big button. Instead, a panel with a `FlowLayout` layout manager with right alignment is used, and the `<Close>` button is added to it. The panel is subsequently added to the south region of the dialog (or more exactly, the content pane of the dialog).

A class `ReportDialog` is shown in Figure 9.127 for such a report dialog.

```
// Resolve classes in java.awt, java.awt.event and javax.swing packages
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

// Definition of class MPFReportFrame
public class ReportDialog extends JDialog implements ActionListener {
    // GUI components
    private JTextArea reportArea = new JTextArea("Report", 15, 60);
    private JButton closeButton = new JButton("Close");

    // Constructor
    public ReportDialog(Frame owner) {
        // Set the dialog title
        super(owner, "MPF Report", true);

        // Build the GUI
        Container contentPane = getContentPane();

        reportArea.setEditable(false);
        Font font = new Font("Monospaced", Font.PLAIN, 14);
        reportArea.setFont(font);
        contentPane.add(reportArea, BorderLayout.CENTER);
        JPanel buttonPanel = new JPanel();
        buttonPanel.setLayout(new FlowLayout(FlowLayout.RIGHT));
        buttonPanel.add(closeButton);
        contentPane.add(buttonPanel, BorderLayout.SOUTH);

        // Register event handler with GUI components
        closeButton.addActionListener(this);

        // Set the default operation if the close button of the frame
        // is clicked
        setDefaultCloseOperation(HIDE_ON_CLOSE);

        // Set the frame to its optimal size
        pack();
    }

    // Clicked if the Close button is clicked
    public void actionPerformed(ActionEvent e) {
        // Hide the frame
        hide();
    }
}
```

```

        // Show the frame with the supplied report
        public void showReport(String report) {
            // Set the content of the text area
            reportArea.setText(report);
            // Show the frame
            show();
        }
    }
}

```

**Figure 9.127** ReportDialog.java

The definition of the `ReportDialog` class should be quite trivial except for the following:

- 1 The typeface of the text area used by the dialog must be set to monospaced, because the report is aligned by columns. To set the typeface of the text area, the following program segment is used:

```

Font font = new Font("Monospaced", Font.PLAIN, 14);
reportArea.setFont(font);

```

The first statement creates a `Font` object with font name `Monospaced`, style as `plain` and font size of 14.

- 2 The panel that contains the <Close> button uses a `FlowLayout` layout manager object with right alignment. Hence the statement to set the layout manager to the panel is:

```

buttonPanel.setLayout(new FlowLayout(FlowLayout.RIGHT));

```

## The staff type selection dialog

The staff type selection dialog as shown in Figure 9.122 uses radio boxes for the user to select a staff type. The dialog uses the default `BorderLayout` object as its layout manager. While building the GUI, a label for the message “Please select the staff type” is added to the north region, and three radio boxes are added to a panel using a `GridLayout` object with three rows. Then it is subsequently added to the center region of the dialog. For the south region of the dialog, a panel is added with the <OK> and <Cancel> buttons.

Only the <OK> and <Cancel> buttons need event handling. Clicking the <OK> and <Cancel> buttons of the dialog sets the attribute `option` to the user selected staff type and `NOT_SELECTED` respectively, and the dialog is hidden by either means. After the dialog is hidden, the method `getOption()` can be called to obtain the user selection.

The event handler method `actionPerformed()` defined in the `StaffTypeDialog` solely determines the source of the event and calls a method for the event accordingly. Those methods are `okButtonClicked()` and `cancelButtonClicked()` for handling the <OK> and <Cancel> buttons if they are clicked respectively.

The definition of the class `StaffTypeDialog` is shown in Figure 9.128. The definition seems to be rather lengthy, but you can study it part by part.

```
// Resolve classes in java.awt, java.awt.event and javax.swing packages
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

// Definition of class StaffTypeDialog
public class StaffTypeDialog extends JDialog implements ActionListener {
    // The literals representing different staff type and if no staff
    // type is selected
    public static final int NOT_SELECTED = 0;
    public static final int CLERK = 1;
    public static final int SALESPERSON = 2;
    public static final int MANAGER = 3;

    // GUI components
    private JLabel label = new JLabel("Please select the staff type:");
    private ButtonGroup optionGroup = new ButtonGroup();
    private JRadioButton clerkOption = new JRadioButton("Clerk");
    private JRadioButton salespersonOption =
        new JRadioButton("Salesperson");
    private JRadioButton managerOption = new JRadioButton("Manager");
    private JButton okButton = new JButton("OK");
    private JButton cancelButton = new JButton("Cancel");

    // The default selection of the user
    private int option = NOT_SELECTED;

    // Constructor
    public StaffTypeDialog(Frame owner) {
        // Set the owner of the dialog, dialog title and set it to be
        // a modal dialog
        super(owner, "Staff type selection", true);

        // Arrange the GUI components
        Container contentPane = getContentPane();
        contentPane.add(label, BorderLayout.NORTH);
        JPanel optionPanel = new JPanel();
        optionPanel.setLayout(new GridLayout(3, 1));
        optionPanel.add(clerkOption);
        optionPanel.add(salespersonOption);
        optionPanel.add(managerOption);
        contentPane.add(optionPanel, BorderLayout.CENTER);
        JPanel buttonPanel = new JPanel();
        buttonPanel.add(okButton);
        buttonPanel.add(cancelButton);
        contentPane.add(buttonPanel, BorderLayout.SOUTH);

        // Associate the radio boxes with a ButtonGroup object
        optionGroup.add(clerkOption);
        optionGroup.add(salespersonOption);
        optionGroup.add(managerOption);
    }
}
```

```

        // Register event handlers
        okButton.addActionListener(this);
        cancelButton.addActionListener(this);

        // Set the dialog to its optimal size
        pack();
    }

    // Method called if the OK or Cancel button is clicked
    public void actionPerformed(ActionEvent e) {
        // Get the event source
        Object eventSource = e.getSource();
        // Execute a method according to the event source
        if (eventSource == okButton) {
            okButtonClicked(e);
        } else if (eventSource == cancelButton) {
            cancelButtonClicked(e);
        }
    }

    // Show the dialog
    public void show() {
        // Set the default user selection to be not selected
        option = NOT_SELECTED;
        // Call the show() method of JDialog to show the dialog
        super.show();
    }

    // Get the option selected by the user
    public int getOption() {
        return option;
    }

    // Method called if the OK button is clicked
    private void okButtonClicked(ActionEvent e) {
        // Set the user option according to the states of the
        // radio boxes
        if (clerkOption.isSelected()) {
            option = CLERK;
        } else if (salespersonOption.isSelected()) {
            option = SALESPERSON;
        } else if (managerOption.isSelected()) {
            option = MANAGER;
        }

        // Hide the dialog if the user has made a selection
        if (option != NOT_SELECTED) {
            hide();
        }
    }

    // Method called if the Cancel button is clicked
    private void cancelButtonClicked(ActionEvent e) {
        // Set the user option to be not selected
        option = NOT_SELECTED;
        // Hide the dialog
        hide();
    }
}

```

Figure 9.128 StaffTypeDialog.java

## The dialogs for obtaining staff data

The payroll software supports three staff types — clerk, salesperson, and manager. The data to be collected for each staff type are different. It is therefore necessary to define three different dialogs. If you study the three dialogs in Figure 9.123 to Figure 9.125 in detail, you will find that they are similar. For example, all dialogs have the <Add> button, <Cancel> button, a list of labels and a list of text fields. Therefore, it is possible to derive a common superclass that has the <Add> button and <Cancel> button. The lists of labels and text fields can be set by the subclasses.

By consolidating the similarities among the staff dialogs, a common superclass `StaffDialog` is defined in Figure 9.129.

```
// Resolve classes in java.awt, java.awt.event and javax.swing package
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

// Definition of class StaffDialog
public abstract class StaffDialog
    extends JDialog
    implements ActionListener {

    // The GUI components
    private JPanel labelPanel = new JPanel();
    private JPanel textFieldPanel = new JPanel();
    private JPanel buttonPanel = new JPanel();

    // The arrays storing the labels and the text fields
    private JLabel[] labels;
    private JTextField[] textfields;

    // The two buttons on the dialog
    private JButton actionButton = new JButton();
    private JButton cancelButton = new JButton("Cancel");

    // The button clicked
    private JButton buttonClicked;

    // Constructor
    public StaffDialog(Frame owner, String frameTitle) {
        // Set the owner of the dialog, the frame title and make
        // it a modal dialog
        super(owner, frameTitle, true);

        // Build the GUI
        Container contentPane = getContentPane();
        contentPane.add(labelPanel, BorderLayout.WEST);
        contentPane.add(textFieldPanel, BorderLayout.CENTER);
        contentPane.add(buttonPanel, BorderLayout.SOUTH);
        buttonPanel.add(actionButton);
        buttonPanel.add(cancelButton);

        // Register event handlers with the buttons
        actionButton.addActionListener(this);
        cancelButton.addActionListener(this);
    }
}
```



```
// Set the labels and text fields of the dialog
public void setEntryForm(JLabel[] labels, JTextField[] textfields) {
    // Add the labels to the label panel
    labelPanel.setLayout(new GridLayout(labels.length, 1));
    for (int i = 0; i < labels.length; i++) {
        labelPanel.add(labels[i]);
    }

    // Add the text fields to the text field panel
    textFieldPanel.setLayout(new GridLayout(textfields.length, 1));
    for (int i = 0; i < textfields.length; i++) {
        textFieldPanel.add(textfields[i]);
    }

    // Set the dialog to its optimal size
    pack();
}

// Add a staff by setting all fields empty and the button title
// to Add, and showing the dialog
public void addStaff() {
    // Setting all fields empty
    setFieldsFromStaff(null);
    // Set the button title to Add
    actionButton.setText("Add");
    // Show the dialog
    show();
}

// Edit a staff by setting the fields according to the
// object supplied, the button title to Update, and show the dialog
public void editStaff(Staff staff) {
    // Set the text fields according to the Staff object provided
    // by calling the method implemented by a subclass
    setFieldsFromStaff(staff);
    // Set the button title to Update
    actionButton.setText("Update");
    // Show the dialog
    show();
}

// Get the Staff object from the text fields
public Staff getStaff() {
    // Check the button clicked
    if (buttonClicked == actionButton) {
        // If the action button is clicked, return a Staff object
        // by calling the method implemented by a subclass
        return getStaffFromFields();
    } else {
        // If the Cancel button is clicked, just return null
        return null;
    }
}
```

```

// Method called if a button is clicked
public void actionPerformed(ActionEvent e) {
    // Store the reference of the button clicked
    if (e.getSource() instanceof JButton) {
        buttonClicked = (JButton) e.getSource();
    }
    // Hide the dialog
    hide();
}

// A method to be implemented by subclass that determine the way to
// convert the text field contents to a Staff object
public abstract Staff getStaffFromFields();

// A method to be implemented by subclass that determine the way to
// set the text field contents according to a supplied Staff object
public abstract void setFieldsFromStaff(Staff staff);
}

```

**Figure 9.129** StaffDialog.java

Besides constructing the GUI, all staff dialogs can create a `Staff` object (such as a `Clerk` object for the dialog for getting clerk data) and set the text field contents according to a supplied `Staff` object (and actually a `Clerk` object for the dialog for handling clerk data). These functionalities are to be implemented by different dialogs of different staff types and are therefore defined to be abstract methods `getStaffFromFields()` and `setFieldsFromStaff()`.

The dialogs can be used for adding data on new staff or editing existing staff data. The text field contents are made empty, and the text of the left button is set to be 'Edit' if the dialog is used to add a new staff member. The text field contents are set according to a supplied `Staff` object, and the text of the left button is set to be 'Update' if the dialog is used to edit an existing staff member's data. These two functionalities are implemented as the methods `addStaff()` and `editStaff()` respectively.

The event handler `actionPerformed()` method stores the button clicked by the user and stores the reference of the button to the attribute `buttonClicked`. No matter which button is clicked, the dialog is hidden. Then, when the `getStaff()` method is called, a `Staff` object that is created by calling the `getStaffFromFields()` is returned if the left button is clicked (referred by the attribute `actionButton`). Otherwise, a value of `null` is returned indicating the <Cancel> button was clicked.

The `setEntryForm()` method is defined to be called by the subclass constructors to set the labels and the text fields by supplying array objects with element types of `JLabel` and `JTextField`.

It is necessary for the concrete subclass dialogs simply to prepare the array objects with element types `JLabel` and `JTextFields`, call the `setEntryForm()` with the array objects, and implement the two abstract methods defined in the abstract superclass. The concrete

subclasses ClerkDialog, SalesPersonDialog and ManagerDialog are written accordingly and are shown in Figure 9.130 to Figure 9.132.

```
// Resolve classes in java.awt and javax.swing packages
import java.awt.*;
import javax.swing.*;

// Definition of class ClerkDialog
public class ClerkDialog extends StaffDialog {
    // GUI Components
    // Labels
    private JLabel nameLabel = new JLabel("Name: ");
    private JLabel basicSalaryLabel = new JLabel("Basic salary: ");
    // An array object maintaining the labels
    private JLabel[] labels = { nameLabel, basicSalaryLabel };
    // Text fields
    private JTextField nameTextField = new JTextField(20);
    private JTextField basicSalaryTextField = new JTextField(10);
    // An array object maintaining the text fields
    private JTextField[] textfields =
        { nameTextField, basicSalaryTextField };

    // Constructor
    public ClerkDialog(Frame owner) {
        // Set the owner of the dialog and the dialog title
        super(owner, "Clerk information");
        // Set the labels and text fields
        setEntryForm(labels, textfields);
    }

    // Create a Clerk object based on the text field contents
    public Staff getStaffFromFields() {
        return new Clerk(
            nameTextField.getText(),
            Double.parseDouble(basicSalaryTextField.getText()));
    }

    // Set the text field contents according to the supplied
    // Staff object (more exactly, a Manager object)
    public void setFieldsFromStaff(Staff staff) {
        if (staff != null) {
            // If the supplied object is not null, cast it to a
            // Manager object and set the text fields
            Clerk clerk = (Clerk) staff;
            nameTextField.setText(clerk.getName());
            basicSalaryTextField.setText("" + clerk.getBasicSalary());
        } else {
            // If no Staff object supplied, just set the text fields
            // empty
            nameTextField.setText(null);
            basicSalaryTextField.setText(null);
        }
    }
}
```

**Figure 9.130** ClerkDialog.java

```
// Resolve classes in java.awt and javax.swing packages
import java.awt.*;
import javax.swing.*;

// Definition of class SalesPersonDialog
public class SalesPersonDialog extends StaffDialog {
    // Labels
    private JLabel nameLabel = new JLabel("Name: ");
    private JLabel basicSalaryLabel = new JLabel("Basic salary: ");
    private JLabel salesVolumeLabel = new JLabel("Sales volume: ");
    private JLabel commissionRateLabel =
        new JLabel("Commission rate (%): ");
    // An array that maintains all JLabel objects
    private JLabel[] labels =
    {
        nameLabel,
        basicSalaryLabel,
        salesVolumeLabel,
        commissionRateLabel };

    // Text fields
    private JTextField nameTextField = new JTextField(20);
    private JTextField basicSalaryTextField = new JTextField(10);
    private JTextField salesVolumeTextField = new JTextField(10);
    private JTextField commissionRateTextField = new JTextField(10);
    // An array that maintains all JTextField objects
    private JTextField[] textfields =
    {
        nameTextField,
        basicSalaryTextField,
        salesVolumeTextField,
        commissionRateTextField };

    // Constructor
    public SalesPersonDialog(Frame owner) {
        // Set the owner of the dialog and the dialog title
        super(owner, "Salesperson information");

        // Build the GUI for labels and textfields
        setEntryForm(labels, textfields);
    }

    // Create a SalesPerson object according to the text field contents
    public Staff getStaffFromFields() {
        return new SalesPerson(
            nameTextField.getText(),
            Double.parseDouble(basicSalaryTextField.getText()),
            Double.parseDouble(salesVolumeTextField.getText()),
            Double.parseDouble(commissionRateTextField.getText())
            / 100.0);
    }
}
```

```

// Set the text field contents according to a Staff (more exactly,
// a SalesPerson) object
public void setFieldsFromStaff(Staff staff) {
    if (staff != null) {
        // If the supplied object is not null, cast it to a
        // SalesPerson object and set the text fields
        SalesPerson salesPerson = (SalesPerson) staff;
        nameTextField.setText(salesPerson.getName());
        basicSalaryTextField.setText(
            "" + salesPerson.getBasicSalary());
        salesVolumeTextField.setText(
            "" + salesPerson.getSalesVolume());
        commissionRateTextField.setText(
            "" + salesPerson.getCommissionRate() * 100.0);
    } else {
        // If no Staff object supplied, just set the text fields
        // empty
        nameTextField.setText(null);
        basicSalaryTextField.setText(null);
        salesVolumeTextField.setText(null);
        commissionRateTextField.setText(null);
    }
}
}

```

**Figure 9.131** SalesPersonDialog.java

```

// Resolve classes in java.awt and javax.swing packages
import java.awt.*;
import javax.swing.*;

// Definition of class ManagerDialog
public class ManagerDialog extends StaffDialog {
    // GUI components
    // Labels
    private JLabel nameLabel = new JLabel("Name: ");
    private JLabel basicSalaryLabel = new JLabel("Basic salary: ");
    private JLabel allowanceLabel = new JLabel("Allowance: ");
    // An array maintaining the labels
    private JLabel[] labels =
        { nameLabel, basicSalaryLabel, allowanceLabel };
    // Text fields
    private JTextField nameTextField = new JTextField(20);
    private JTextField basicSalaryTextField = new JTextField(10);
    private JTextField allowanceTextField = new JTextField(10);
    // An array maintaining the text fields
    private JTextField[] textfields =
        { nameTextField, basicSalaryTextField, allowanceTextField };

    // Constructor
    public ManagerDialog(Frame owner) {
        // Set the owner of the dialog and dialog title
        super(owner, "Manager information");
        // Set the labels and text fields of the dialog
        setEntryForm(labels, textfields);
    }
}

```

```

// Create a Manager object based on the text field contents
public Staff getStaffFromFields() {
    return new Manager(
        nameTextField.getText(),
        Double.parseDouble(basicSalaryTextField.getText()),
        Double.parseDouble(allowanceTextField.getText()));
}

// Set the text field contents according to the supplied
// Staff object (more exactly, a Manager object)
public void setFieldsFromStaff(Staff staff) {
    if (staff != null) {
        // If the supplied object is not null, cast it to a
        // Manager object and set the text fields
        Manager manager = (Manager) staff;
        nameTextField.setText(manager.getName());
        basicSalaryTextField.setText("" + manager.getBasicSalary());
        allowanceTextField.setText("" + manager.getAllowance());
    } else {
        // If no Staff object supplied, just set the text fields
        // empty
        nameTextField.setText(null);
        basicSalaryTextField.setText(null);
        allowanceTextField.setText(null);
    }
}
}

```

**Figure 9.132** ManagerDialog.java

By studying the definitions of the classes `ClerkDialog`, `SalesPersonDialog` and `ManagerDialog`, you can see that they are implemented using the abstract superclass and concrete subclasses that we discussed in *Unit 7*. The common abstract superclass defines all common attributes and behaviours of all subclasses. The functionalities to be implemented by specific subclasses are defined to be abstract methods and are left to the subclasses to implement. Therefore, the subclasses are greatly simplified, and no redundant source codes are required to be replicated in each subclass. This is also one of the ways object-oriented programming improves the productivity of software developers.

## The main frame

Here comes the most complex class definition of all. The greatest number of buttons is embedded in the frame and defines a menu bar. Therefore, attributes are defined at least for buttons, menu bar, menus and menu items. Furthermore, various dialogs are needed for obtaining user entries. Attributes are defined for report dialog, staff type selection dialog and different staff type data entry dialog. It also needs the helper classes that are obtained from previous units or modified based on them, including classes `StaffFileHandler` and `StaffArrayHandler`. Finally, and most importantly, the frame has to maintain the staff array, the current index of the current `Staff` object maintained by the array object, and the current staff file name.

The main frame is modelled by the `PayrollFrame` defined in Figure 9.133. It is the longest definition in this course up to now. Please do not be scared by the length. If you read it carefully, it follows the usual template for defining a frame. For example, the class defines attributes for embedded GUI components, all helper objects, and the data to be handled. The attribute declarations are followed by the constructor, which arranges the GUI components and registers event-handler objects with the GUI components. The frame handles only Action and Window event categories, and the methods `actionPerformed()` and `windowClosing()` are defined. The remaining event-handler methods defined in the `WindowListener` interface are implemented as dummy methods that contain no statements. Finally, there are some utility methods that are to be called by other methods.

The `actionPerformed()` method determines the source of the event and calls the corresponding method accordingly. For example, if the <Open> menu item is selected, the method `openMenuItemSelected()` is eventually called. The methods are written to be self-explanatory, so you can figure out their functionalities by reading the comments.

```
// Resolve classes in java.awt, java.awt.event, java.io
// and javax.swing packages
import java.awt.*;
import java.awt.event.*;
import java.io.IOException;
import javax.swing.*;

// Definition of class PayrollFrame
public class PayrollFrame
    extends JFrame
    implements ActionListener, WindowListener {

    // GUI components for the menu bar
    private JMenuBar menuBar = new JMenuBar();
    private JMenu fileMenu = new JMenu("File");
    private JMenu reportMenu = new JMenu("Report");
    private JMenu helpMenu = new JMenu("Help");
    private JMenuItem newMenuItem = new JMenuItem("New");
    private JMenuItem openMenuItem = new JMenuItem("Open...");
    private JMenuItem saveMenuItem = new JMenuItem("Save");
    private JMenuItem saveAsMenuItem = new JMenuItem("Save As...");
    private JMenuItem exitMenuItem = new JMenuItem("Exit");
    private JMenuItem reportMenuItem = new JMenuItem("MPF Report");
    private JMenuItem aboutMenuItem =
        new JMenuItem("About Payroll Software");

    // GUI components for the buttons, text field and label
    private JLabel nameLabel = new JLabel("Name: ");
    private JButton firstButton = new JButton("First");
    private JButton previousButton = new JButton("Previous");
    private JButton nextButton = new JButton("Next");
    private JButton lastButton = new JButton("Last");
    private JTextField nameField = new JTextField(10);
    private JLabel positionLabel =
```

```

        new JLabel("      ", SwingConstants.CENTER);
private JButton addButton = new JButton("Add");
private JButton editButton = new JButton("Edit");
private JButton deleteButton = new JButton("Delete");

// The dialogs to be used
private ReportDialog reportFrame = new ReportDialog(this);
private StaffTypeDialog staffTypeFrame = new StaffTypeDialog(this);
private ClerkDialog clerkDialog = new ClerkDialog(this);
private SalesPersonDialog salesPersonDialog =
    new SalesPersonDialog(this);
private ManagerDialog managerDialog = new ManagerDialog(this);
private JFileChooser chooser = new JFileChooser();

// The staff array object
private Staff[] staffArray = new Staff[1000];
// The index indicating the current Staff object maintained by the
// staff array
private int staffNumber;

// The staff array handler
private StaffArrayHandler arrayHandler =
    new StaffArrayHandler(staffArray);
// The file handler for performing input/output operations
private StaffFileHandler fileHandler = new StaffFileHandler();
// The current file name
private String staffFileName;

// Constructor
public PayrollFrame() {
    // Set the frame title
    super("Payroll Calculation");

    // Build the menu bar
    setJMenuBar(menuBar);
    menuBar.add(fileMenu);
    menuBar.add(reportMenu);
    menuBar.add(helpMenu);
    fileMenu.add(newMenuItem);
    fileMenu.add(openMenuItem);
    fileMenu.add(saveMenuItem);
    fileMenu.add(saveAsMenuItem);
    fileMenu.addSeparator();
    fileMenu.add(exitMenuItem);
    reportMenu.add(reportMenuItem);
    helpMenu.add(aboutMenuItem);

    // Build other GUI components
    Container contentPane = getContentPane();
    contentPane.add(nameLabel, BorderLayout.WEST);
    contentPane.add(nameField, BorderLayout.CENTER);
    JPanel controlPanel = new JPanel();
    controlPanel.add(editButton);
    controlPanel.add(deleteButton);
    contentPane.add(controlPanel, BorderLayout.EAST);
    JPanel navigationPanel = new JPanel();

```



```

        navigationPanel.add(firstButton);
        navigationPanel.add(previousButton);
        navigationPanel.add(positionLabel);
        navigationPanel.add(nextButton);
        navigationPanel.add(lastButton);
        navigationPanel.add(addButton);
        contentPane.add(navigationPanel, BorderLayout.SOUTH);

        // The name field is set to be non-editable
        nameField.setEditable(false);

        // Register event handlers to the menu items and buttons
        newMenuItem.addActionListener(this);
        openMenuItem.addActionListener(this);
        saveMenuItem.addActionListener(this);
        saveAsMenuItem.addActionListener(this);
        exitMenuItem.addActionListener(this);
        reportMenuItem.addActionListener(this);
        aboutMenuItem.addActionListener(this);
        editButton.addActionListener(this);
        deleteButton.addActionListener(this);
        firstButton.addActionListener(this);
        previousButton.addActionListener(this);
        nextButton.addActionListener(this);
        lastButton.addActionListener(this);
        addButton.addActionListener(this);
        // Register event handlers to this frame
        addWindowListener(this);

        // Set the default operation when the close button of the frame
        // is clicked
        setDefaultCloseOperation(DO_NOTHING_ON_CLOSE);

        // Set the frame to optimal size
        pack();
    }

    // Method called if any menu item is selected or button is clicked
    public void actionPerformed(ActionEvent e) {
        // Get the source of the event
        Object eventSource = e.getSource();

        // Compare the event source with all GUI components that
        // registers the object itself as listener. Once matched, call
        // the method to handle the event.
        if (eventSource == newMenuItem) {
            newMenuItemSelected(e);
        } else if (eventSource == openMenuItem) {
            openMenuItemSelected(e);
        } else if (eventSource == saveMenuItem) {
            saveMenuItemSelected(e);
        } else if (eventSource == saveAsMenuItem) {
            saveAsMenuItemSelected(e);
        } else if (eventSource == exitMenuItem) {
            exitMenuItemSelected(e);
        }
    }

```

```

        } else if (eventSource == reportMenuItem) {
            reportMenuItemSelected(e);
        } else if (eventSource == aboutMenuItem) {
            aboutMenuItemSelected(e);
        } else if (eventSource == editButton) {
            editButtonClicked(e);
        } else if (eventSource == deleteButton) {
            deleteButtonClicked(e);
        } else if (eventSource == firstButton) {
            firstButtonClicked(e);
        } else if (eventSource == previousButton) {
            previousButtonClicked(e);
        } else if (eventSource == nextButton) {
            nextButtonClicked(e);
        } else if (eventSource == lastButton) {
            lastButtonClicked(e);
        } else if (eventSource == addButton) {
            addButtonClicked(e);
        }
    }

    // Method called if the Close button of the frame is clicked
    public void windowClosing(WindowEvent we) {
        // Confirm the user whether to terminate the program
        confirmExit();
    }

    // The methods corresponding to events that do not need handling
    public void windowActivated(WindowEvent we) {}
    public void windowClosed(WindowEvent we) {}
    public void windowDeactivated(WindowEvent we) {}
    public void windowDeiconified(WindowEvent we) {}
    public void windowIconified(WindowEvent we) {}
    public void windowOpened(WindowEvent we) {}

    // Method called if the New menu item is selected
    private void newMenuItemSelected(ActionEvent e) {
        // Check the number of Staff object maintained by the program
        if (arrayHandler.getStaffTotal() > 0) {
            // If there are Staff objects, prompt the user to see
            // whether it is necessary to save the existing objects
            int reply =
                JOptionPane.showConfirmDialog(
                    this,
                    "Do you want to save the existing staff data?",
                    "Save staff data",
                    JOptionPane.YES_NO_CANCEL_OPTION);

            // Save the objects if necessary. Anyway,
            if (reply == JOptionPane.YES_OPTION) {
                saveStaffArray();
            }

            // If the option is not to cancel the operation,
            // create a new Staff array and array handler, and reset
            // the position label

```

```

        if (reply != JOptionPane.CANCEL_OPTION) {
            staffArray = new Staff[1000];
            arrayHandler = new StaffArrayHandler(staffArray);
            gotoStaffNumber(-1);
        }
    }

    // Method called if the Open menu item is selected
    private void openMenuItemSelected(ActionEvent e) {
        int reply = JOptionPane.YES_OPTION;
        // Check the number of Staff object maintained by the program
        if (arrayHandler.getStaffTotal() > 0) {
            // If there are Staff objects, prompt the user to see
            // whether it is necessary to save the existing objects
            reply =
                JOptionPane.showConfirmDialog(
                    this,
                    "Do you want to save the existing staff data?",
                    "Save staff data",
                    JOptionPane.YES_NO_CANCEL_OPTION);

            // Save the objects if necessary. Anyway,
            if (reply == JOptionPane.YES_OPTION) {
                saveStaffArray();
            }
        }

        // If the user did not click the Cancel button
        if (reply != JOptionPane.CANCEL_OPTION) {
            // Use a JFileChooser to select a file
            reply = chooser.showOpenDialog(this);

            // If the user select a file
            if (reply == JFileChooser.APPROVE_OPTION) {
                // Read the Staff array object from the file and copy
                // the Staff objects to a new Staff array
                staffFileName = chooser.getSelectedFile().getPath();
                try {
                    Staff[] tempArray =
                        fileHandler.readStaffData(staffFileName);
                    staffArray = new Staff[1000];
                    System.arraycopy(
                        tempArray,
                        0,
                        staffArray,
                        0,
                        tempArray.length);
                    arrayHandler = new StaffArrayHandler(staffArray);
                    // Show the first Staff object
                    gotoStaffNumber(0);
                } catch (IOException ioe) {
                    JOptionPane.showMessageDialog(
                        this,
                        "Cannot read staff file.\nReason: "
                        + ioe.getMessage());
                }
            }
        }
    }

```

```

        }
    }
}

// Method called if the Save menu item is selected
private void saveMenuItemSelected(ActionEvent e) {
    // Save the existing Staff array to the current
    // staff file name
    saveStaffArray();
}

// Method called if the Save As menu item is selected
private void saveAsMenuItemSelected(ActionEvent e) {
    // Store the original file name temporarily
    String originalFileName = staffFileName;
    // Unset the staff file name so that the user is prompted a
    // file selection dialog to select a file name
    staffFileName = null;
    // Try to save the file
    saveStaffArray();
    // Check if the file is saved with another file name or not. If
    // the file is not saved with another name, restore the
    // original name
    if (staffFileName == null) {
        staffFileName = originalFileName;
    }
}

// Method called if the Exit menu item is selected
private void exitMenuItemSelected(ActionEvent e) {
    // Confirm whether the user want to terminate the program
    confirmExit();
}

// Method called if the Report menu item is selected
private void reportMenuItemSelected(ActionEvent e) {
    // Create a Staff array with array size equals the
    // total number of Staff objects
    Staff[] allStaff = new Staff[arrayHandler.getStaffTotal()];
    // Copy the Staff array from the processing one
    // to the new one
    System.arraycopy(staffArray, 0, allStaff, 0, allStaff.length);
    // Create a PayrollCalculator object
    PayrollCalculator calculator = new PayrollCalculator(allStaff);
    // Get the report
    String report = calculator.getReport();
    // Show the report frame
    reportFrame.showReport(report);
}

// Method called if the About menu item is selected
private void aboutMenuItemSelected(ActionEvent e) {
    // Show a message dialog
    JOptionPane.showMessageDialog(
        this,

```

```

        "Payroll Software\n"
        + "version 1.0\n"
        + "By Open University of Hong Kong");
    }

    // Method called if the Edit button is clicked
    private void editButtonClicked(ActionEvent e) {
        // Get the reference of the Staff object to be edited
        Staff thisStaff = staffArray[staffNumber];
        if (thisStaff != null) {
            // Edit the Staff object according to its real class by
            // showing the suitable dialog and the returned Staff
            // object is referred by the editedStaff variable
            Staff editedStaff = null;
            if (thisStaff instanceof Clerk) {
                clerkDialog.editStaff(thisStaff);
                editedStaff = clerkDialog.getStaff();
            } else if (thisStaff instanceof SalesPerson) {
                salesPersonDialog.editStaff(thisStaff);
                editedStaff = salesPersonDialog.getStaff();
            } else if (thisStaff instanceof Manager) {
                managerDialog.editStaff(thisStaff);
                editedStaff = managerDialog.getStaff();
            }

            // If there is a Staff object returned, the Update button
            // of the dialog is clicked and the returned Staff object
            // replace the original one in the Staff array
            if (editedStaff != null) {
                staffArray[staffNumber] = editedStaff;
            }
        }
    }

    // Method called if the Delete button is clicked
    private void deleteButtonClicked(ActionEvent e) {
        // Get the staff name from the name text field
        String staffName = nameField.getText();
        // If the name is not null
        if (staffName != null) {
            // Remove a Staff object from the staff array by supplying
            // the staff name
            arrayHandler.removeStaff(staffName);
            // Refresh the name text field and position label
            gotoStaffNumber(staffNumber);
        }
    }

    // Method called if the First button is clicked
    private void firstButtonClicked(ActionEvent e) {
        // Go to the first record
        gotoStaffNumber(0);
    }
}

```

```

// Method called if the Previous button is clicked
private void previousButtonClicked(ActionEvent e) {
    // Go to the previous record
    if (staffNumber > 0) {
        gotoStaffNumber(staffNumber - 1);
    }
}

// Method called if the Next button is clicked
private void nextButtonClicked(ActionEvent e) {
    // Go to the next record
    if (staffArray != null && staffNumber < staffArray.length) {
        gotoStaffNumber(staffNumber + 1);
    }
}

// Method called if the Last button is clicked
private void lastButtonClicked(ActionEvent e) {
    // Go to the last record
    if (staffArray != null) {
        gotoStaffNumber(staffArray.length - 1);
    }
}

// Method called if the Add button is clicked
private void addButtonClicked(ActionEvent e) {
    // Show the dialog for the user to choose the staff type
    staffTypeFrame.show();
    // Declare a variable to store the new Staff object
    Staff newStaff = null;
    // Show the suitable dialog according to the Staff type
    // selected
    switch (staffTypeFrame.getOption()) {
        case StaffTypeDialog.CLERK :
            clerkDialog.addStaff();
            newStaff = clerkDialog.getStaff();
            break;
        case StaffTypeDialog.SALESPERSON :
            salesPersonDialog.addStaff();
            newStaff = salesPersonDialog.getStaff();
            break;
        case StaffTypeDialog.MANAGER :
            managerDialog.addStaff();
            newStaff = managerDialog.getStaff();
            break;
    }

    // Checked if a new Staff object is returned
    if (newStaff != null) {
        // Add the Staff object
        arrayHandler.addStaff(newStaff);
        // Go to the last record, which is the newly added Staff
        // object
        gotoStaffNumber(arrayHandler.getStaffTotal() - 1);
    }
}

```

```

    }
}

// Show the desired Staff object and the position label according
// to the number supplied
private void gotoStaffNumber(int toNumber) {
    // Check if there are staff objects
    if (toNumber == -1 || arrayHandler.getStaffTotal() == 0) {
        // Set the name field and position empty
        nameField.setText(null);
        positionLabel.setText(null);
    } else {
        // Store the current staff number
        staffNumber = toNumber;
        if (staffArray != null) {
            // Adjust the record number to be shown if necessary
            if (staffNumber < 0) {
                staffNumber = 0;
            }
            if (staffNumber >= arrayHandler.getStaffTotal()) {
                staffNumber = arrayHandler.getStaffTotal() - 1;
            }
            // Set the name text field
            nameField.setText(staffArray[staffNumber].getName());
            // Set the position label
            positionLabel.setText(
                (staffNumber + 1)
                + "/"
                + arrayHandler.getStaffTotal());
        }
    }
}

// Save the staff array to a file
private void saveStaffArray() {
    if (staffArray != null) {
        // If the staff array is referring to a Staff array object
        try {
            if (staffFileName == null) {
                // If there is no default staff file name, prompt
                // the user for the file name
                int reply = chooser.showSaveDialog(this);
                if (reply == JFileChooser.APPROVE_OPTION) {
                    staffFileName =
                        chooser.getSelectedFile().getPath();
                    fileHandler.writeStaffData(
                        staffFileName,
                        staffArray);
                }
            }
        }
    }
}

```

```

        } else {
            // If there is a default staff file name, save the
            // staff array
            fileHandler.writeStaffData(
                staffFileName,
                staffArray);
        }
    } catch (IOException e) {
        JOptionPane.showMessageDialog(
            this,
            "Cannot write staff file.");
    }
}

// Confirm the user whether to terminate the program
private void confirmExit() {
    if (arrayHandler.getStaffTotal() > 0) {
        // Prompt the user to confirm whether to terminate the
        // program
        int reply =
            JOptionPane.showConfirmDialog(
                this,
                "Do you want to save the staff data?");
        // Depend on the response of the user, save the staff
        // array and/or terminate the program
        switch (reply) {
            case JOptionPane.YES_OPTION :
                saveStaffArray();
            case JOptionPane.NO_OPTION :
                System.exit(0);
        }
    } else {
        // If there is no Staff objects, just terminate
        System.exit(0);
    }
}
}

```

**Figure 9.133** PayrollFrame.java

### The class for payroll calculation

In *Unit 8*, the payroll calculation class shows the report by the standard output. As the report to be shown by this program is presented in a text area, it is necessary to obtain a `String` object with contents of the payroll report. The `PayrollCalculator` class is written as shown in Figure 9.134 based on the one discussed in *Unit 8*.



```

// Definition of class PayrollCalculator
public class PayrollCalculator {
    // Attribute
    private Staff[] staffList;    // The staff list to be processed

    // Constructor
    public PayrollCalculator(Staff[] staffList) {
        this.staffList = staffList;
    }

    public String getReport() {
        // Declare and initialize running totals
        double totalSalary = 0.0;
        double totalMPFByStaff = 0.0;
        double totalMPFByCompany = 0.0;

        String report = "";

        // Show the listing title
        report += "\n        Payroll (MPF) Report\n";
        report += "Staff\tRaw\tNet\tMPF by\tMPF by\n";
        report += "Name\tSalary\tSalary\tStaff\tCompany\n";
        report += "-----\n";

        // Iterate each staff to show his/her details
        for (int i=0; i < staffList.length; i++) {
            // Get the staff payroll details
            double salary = staffList[i].findSalary();
            double netSalary = staffList[i].findNetSalary();
            double mpfByStaff = staffList[i].findMPFByStaff();
            double mpfByCompany = staffList[i].findMPFByCompany();

            // Show the details
            report +=
                staffList[i].getName() +
                "\t" + salary +
                "\t" + netSalary +
                "\t" + mpfByStaff +
                "\t" + mpfByCompany + "\n";

            // Update the running totals
            totalSalary += netSalary;
            totalMPFByStaff += mpfByStaff;
            totalMPFByCompany += mpfByCompany;
        }

        // Show the grand totals
        report += "-----\n";
        report +=
            "Total" +
            "\t\t" + totalSalary +
            "\t" + totalMPFByStaff +
            "\t" + totalMPFByCompany + "\n";

        return report;
    }
}

```

Figure 9.134 PayrollCalculator.java

The `getReport()` method performs MPF and payroll calculations. The lines for the reports are concatenated to form a resultant `String` object to be returned.

### The driver program of the payroll software application

All necessary classes are defined. The last one to be written is the driver program. It is written as the `PayrollSoftware` class as defined in Figure 9.135.

```
// Definition of class PayrollSoftware
public class PayrollSoftware {

    // Main executive method
    public static void main(String args[]) {
        // Create a PayrollFrame object
        PayrollFrame frame = new PayrollFrame();

        // Request the CommandProcessor to show main menu
        // for handling user instructions
        frame.show();
    }
}
```

**Figure 9.135** `PayrollSoftware.java`

## Conclusion

The payroll calculation software is a rather complex program. It is built based on the knowledge you have acquired so far in the course, such as inheritance for defining the classes `Staff`, `Clerk`, `SalesPerson` and `Manager`, and the classes `StaffDialog`, `ClerkDialog`, `SalesPersonDialog` and `ManagerDialog`, input/output operations for reading/writing staff data file, and building GUI for the different frames and dialogs.

A software application may seem to be complicated at first glance. However, by following the software development cycle introduced in *Unit 2*, individual objects and hence the classes to be written are identified. Then, you can investigate whether there are ‘is a’ and ‘has a’ relationships among them. Finally, you can write the classes one by one and the resultant software application can be obtained.

## References

Java Foundation Classes (JFC), at <http://java.sun.com/products/jfc/>

Java Technology: The early years, at  
<http://java.sun.com/features/1998/05/birthday.html>

# Suggested answers to self-test questions

## *Self-test 9.1*

### **FrameWithLabel1.java**

```
// Resolve class Container
import java.awt.*;
// Resolve class JFrame and JLabel
import javax.swing.*;

// Definition of class FrameWithLabel1
public class FrameWithLabel1 extends JFrame {
    // Attribute
    private JLabel label;    // The label that the frame possesses

    // Constructor
    public FrameWithLabel1(String frameTitle, String labelTitle) {
        // Call super class constructor with a title
        super(frameTitle);

        // Create a JLabel
        label = new JLabel(labelTitle);

        // Get the content pane
        Container contentPane = getContentPane();

        // Add the JLabel object to the content pane
        contentPane.add(label);

        // Set when the close button is clicked, the application exits
        setDefaultCloseOperation(EXIT_ON_CLOSE);

        // Reorganize the embedded components
        pack();
    }
}
```

### **TestFrameWithLabel1.java**

```
// Definition of class TestFrameWithLabel1
public class TestFrameWithLabel1 {

    // Main executive method
    public static void main(String args[]) {
        if (args.length < 2) {
            System.out.println(
                "Usage: java TestFrameWithLabel1 " +
                "<frame title> <label title>");
        }
        else {
            // Create a FrameWithLabel1 object
```

```

        FrameWithLabel1 frame = new FrameWithLabel1(
            args[0], args[1]);

        // Show it on the screen
        frame.show();
    }
}

```

## Self-test 9.2

### 1 GreetingFrame1.java

```

// Resolve classes in java.awt and javax.swing packages
import java.awt.*;
import javax.swing.*;

// Definition of class GreetingFrame1
public class GreetingFrame1 extends JFrame {
    // GUI components
    private JButton morningButton = new JButton("Good Morning");
    private JButton afternoonButton = new JButton("Good Afternoon");
    private JButton eveningButton = new JButton("Good Evening");

    // Constructors
    public GreetingFrame1() {
        // Set the frame title
        super("Greeting Frame");

        // Arrange the GUI components
        Container contentPane = getContentPane();
        contentPane.add(morningButton, BorderLayout.NORTH);
        contentPane.add(afternoonButton, BorderLayout.CENTER);
        contentPane.add(eveningButton, BorderLayout.SOUTH);

        // Set the default operation when the close button is clicked
        setDefaultCloseOperation(EXIT_ON_CLOSE);

        // Set the GUI to its optimal size
        pack();
    }
}

```

### TestGreetingFrame1.java

```

// Definition of class TestGreetingFrame1
public class TestGreetingFrame1 {

    // Main executive method
    public static void main(String args[]) {
        // Create a GreetingFrame1 object
        GreetingFrame1 frame = new GreetingFrame1();

        // Show it on the screen
        frame.show();
    }
}

```

### GreetingFrame2.java

```
// Resolve classes in java.awt and javax.swing packages
import java.awt.*;
import javax.swing.*;

// Definition of class GreetingFrame2
public class GreetingFrame2 extends JFrame {
    // GUI components
    private JButton morningButton = new JButton("Good Morning");
    private JButton afternoonButton = new JButton("Good Afternoon");
    private JButton eveningButton = new JButton("Good Evening");
    // Constructors
    public GreetingFrame2() {
        // Set the frame title
        super("Greeting Frame");

        // Arrange the GUI components
        Container contentPane = getContentPane();
        // Set the layout manager to a GridLayout object
        contentPane.setLayout(new GridLayout(3, 1));
        contentPane.add(morningButton);
        contentPane.add(afternoonButton);
        contentPane.add(eveningButton);

        // Set the default operation when the close button is clicked
        setDefaultCloseOperation(EXIT_ON_CLOSE);

        // Set the GUI to its optimal size
        pack();
    }
}
```

### TestGreetingFrame2.java

```
// Definition of class TestGreetingFrame2
public class TestGreetingFrame2 {

    // Main executive method
    public static void main(String args[]) {
        // Create a GreetingFrame2 object
        GreetingFrame2 frame = new GreetingFrame2();

        // Show it on the screen
        frame.show();
    }
}
```

The appearance of the GreetingFrame1 and GreetingFrame2 are the same when they are shown on the screen. However, when the frames are resized, the second button of the GreetingFrame1 is resized, whereas all buttons of the GreetingFrame2 object are resized.

## 2 KeyPadFrame.java

```
// Resolve classes in java.awt and javax.swing packages
import java.awt.*;
import javax.swing.*;

// Definition of the class KeyPadFrame
public class KeyPadFrame extends JFrame {
    // A two dimensional array maintaining the button titles
    private String keyTitles[][] = {
        {"7", "8", "9"},
        {"4", "5", "6"},
        {"1", "2", "3"},
        {"0", ".", "="}
    };

    // A two dimensional array maintaining the JButton objects
    private JButton[][] buttons;

    // Constructor
    public KeyPadFrame() {
        // Set the frame title
        super("KeyPadFrame");

        // Arrange the GUI components
        Container contentPane = getContentPane();
        // Create a GridLayout object based on the dimensions
        // of the two dimensional array
        contentPane.setLayout(
            new GridLayout(keyTitles.length, keyTitles[0].length));
        // Create an array to maintain the JButton object
        buttons = new JButton[keyTitles.length][keyTitles[0].length];
        // Create JButton objects and are added to the content pane
        // one by one
        for (int i=0; i < keyTitles.length; i++) {
            for (int j=0; j < keyTitles[i].length; j++) {
                buttons[i][j] = new JButton(keyTitles[i][j]);
                contentPane.add(buttons[i][j]);
            }
        }

        // Set the default operation when the Close button is clicked
        setDefaultCloseOperation(EXIT_ON_CLOSE);

        // Set the GUI to its optimal size
        pack();
    }
}
```

## TestKeyPadFrame.java

```
// Definition of class TestKeyPadFrame
public class TestKeyPadFrame {

    // Main executive method
    public static void main(String args[]) {
        // Create a KeyPadFrame object
        KeyPadFrame frame = new KeyPadFrame();

        // Show it on the screen
        frame.show();
    }
}
```

### *Self-test 9.3*

#### **CategorySelectionFrame.java**

```
// Resolve classes in java.awt and javax.swing packages
import java.awt.*;
import javax.swing.*;

// Definition of class CategorySelectionFrame
public class CategorySelectionFrame extends JFrame {
    // GUI components
    private ButtonGroup optionGroup = new ButtonGroup();
    private JRadioButton maleChild = new JRadioButton();
    private JRadioButton femaleChild = new JRadioButton();
    private JRadioButton maleAdult = new JRadioButton();
    private JRadioButton femaleAdult = new JRadioButton();
    private JRadioButton maleSenior = new JRadioButton();
    private JRadioButton femaleSenior = new JRadioButton();

    // Constructor
    public CategorySelectionFrame() {
        // Set frame title
        super("Category Selection");

        // Arrange GUI components
        Container contentPane = getContentPane();
        contentPane.setLayout(new GridLayout(4, 3));
        contentPane.add(new JLabel());
        contentPane.add(new JLabel("Male"));
        contentPane.add(new JLabel("Female"));
        contentPane.add(new JLabel("Child"));
        contentPane.add(maleChild);
        contentPane.add(femaleChild);
        contentPane.add(new JLabel("Adult"));
        contentPane.add(maleAdult);
        contentPane.add(femaleAdult);
        contentPane.add(new JLabel("Senior"));
        contentPane.add(maleSenior);
        contentPane.add(femaleSenior);

        // Associate all radio boxes with a ButtonGroup object
        optionGroup.add(maleChild);
        optionGroup.add(femaleChild);
        optionGroup.add(maleAdult);
        optionGroup.add(femaleAdult);
        optionGroup.add(maleSenior);
        optionGroup.add(femaleSenior);

        // Set the default operation when the Close button is clicked
        setDefaultCloseOperation(EXIT_ON_CLOSE);

        // Set the GUI to its optimal size
        pack();
    }
}
```



**TestCategorySelectionFrame.java**

```
// Definition of class TestCategorySelectionFrame
public class TestCategorySelectionFrame {

    // Main executive method
    public static void main(String args[]) {
        // Create a CategorySelectionFrame object
        CategorySelectionFrame frame = new CategorySelectionFrame();

        // Show it on the screen
        frame.show();
    }
}
```

**MultiplicationTable.java**

```
// Resolve classes in java.awt and javax.swing packages
import java.awt.*;
import javax.swing.*;

// Definition of class MultiplicationTable
public class MultiplicationTable extends JFrame {

    public MultiplicationTable() {
        // Set the frame title
        super("Multiplication Table");

        // Get the content pane of the frame and set the layout to be
        // GridLayout with 10 rows and 10 columns
        Container contentPane = getContentPane();
        contentPane.setLayout(new GridLayout(10, 10));

        // Create the GUI components and add them to the content pane
        contentPane.add(new JLabel());
        for (int i=1; i < 10; i++) {
            contentPane.add(new JLabel("" + i, SwingConstants.CENTER));
        }
        for (int i=1; i < 10; i++) {
            contentPane.add(new JLabel("" + i, SwingConstants.CENTER));
            for (int j=1; j < 10; j++) {
                contentPane.add(new JButton("" + i * j));
            }
        }

        // Set the default operation when the close button of the frame
        // is clicked
        setDefaultCloseOperation(EXIT_ON_CLOSE);

        // Set the GUI to its optimal size
        pack();
    }
}
```

**TestMultiplicationTable.java**

```
// Definition of class TestMultiplicationTable
public class TestMultiplicationTable {

    // Main executive method
    public static void main(String args[]) {
        // Create an MultiplicationTable object
        MultiplicationTable frame = new MultiplicationTable();

        // Show it on the screen
        frame.show();
    }
}
```

**Self-test 9.4****1 MessageFrame.java**

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

// Definition of class MessageFrame
public class MessageFrame extends JFrame {
    // Attributes
    private JLabel messageLabel = new JLabel();
    private JButton okButton = new JButton("Yes");
    private JButton cancelButton = new JButton("No");

    // Constructor
    public MessageFrame(String message) {
        // Provide a String to the superclass constructor
        // to set the frame title
        super("Message");

        // Get the content pane of the frame for adding GUI components
        Container contentPane = getContentPane();

        // Create a button panel
        JPanel buttonPanel = new JPanel();
        buttonPanel.setLayout(new GridLayout(1, 2));
        buttonPanel.add(okButton);
        buttonPanel.add(cancelButton);

        // Add the label and the button panel to the content pane
        contentPane.add(messageLabel, BorderLayout.CENTER);
        contentPane.add(buttonPanel, BorderLayout.SOUTH);

        // Set the label text according to the constructor parameter
        messageLabel.setText(message);

        // Set the default behaviour of clicking the close button
        setDefaultCloseOperation(EXIT_ON_CLOSE);

        // Set the frame to its optimal size
        pack();
    }
}
```

**TestMessageFrame.java**

```
// Definition of class TestMessageFrame
public class TestMessageFrame {

    // Main executive method
    public static void main(String args[]) {
        // Create a MessageFrame object
        MessageFrame frame = new MessageFrame("Are you OK?");

        // Show the object on the screen
        frame.show();
    }
}
```

**2 TicTacToeFrame.java**

```
// Resolve classes in java.awt and javax.swing packages
import java.awt.*;
import javax.swing.*;

// Definition of class TicTacToeFrame
public class TicTacToeFrame extends JFrame {
    // The prefix of the status bar message
    private static final String STATUS_PREFIX = "Status: ";
    // GUI components
    private JButton[][] buttons = new JButton[3][3];
    private JButton newGameButton = new JButton("New Game");
    private JButton exitButton = new JButton("Exit");
    private JLabel statusBar = new JLabel(STATUS_PREFIX);

    // Constructor
    public TicTacToeFrame() {
        // Set the frame dialog
        super("Tic Tac Toe");

        // Prepare a button panel
        JPanel actionPanel = new JPanel();
        actionPanel.add(newGameButton);
        actionPanel.add(exitButton);
        // Prepare a tic-tac-toe panel
        JPanel buttonPanel = new JPanel();
        buttonPanel.setLayout(new GridLayout(3, 3));
        for (int i=0; i < buttons.length; i++) {
            for (int j=0; j < buttons[0].length; j++) {
                buttons[i][j] = new JButton(" ");
                buttonPanel.add(buttons[i][j]);
            }
        }
        // Add the button panel, tic-tac-toe panel and the status
        // bar to the content pane
        Container contentPane = getContentPane();
        contentPane.add(actionPanel, BorderLayout.NORTH);
        contentPane.add(buttonPanel, BorderLayout.CENTER);
        contentPane.add(statusBar, BorderLayout.SOUTH);
    }
}
```

```
        // Set the default operation when the Close button is clicked
        setDefaultCloseOperation(EXIT_ON_CLOSE);

        // Set the GUI to its optimal size
        pack();
    }
}
```

### **TestTicTacToeFrame.java**

```
// Definition of class TestTicTacToeFrame
public class TestTicTacToeFrame {

    // Main executive method
    public static void main(String args[]) {
        // Create a TicTacToeFrame object
        TicTacToeFrame frame = new TicTacToeFrame();

        // Show the object on the screen
        frame.show();
    }
}
```

- 3 As the layouts of the dialogs are similar, it is preferable to define a common superclass, the `StaffFrame` class, which arranges the common GUI components. Furthermore, specific GUI components of the dialogs are prepared by the subclasses. A method `setEntryForm()` method is defined by the superclass to set the specific GUI components.

### **StaffFrame.java**

```
// Resolve classes in java.awt, java.awt.event and javax.swing package
import java.awt.*;
import javax.swing.*;

// Definition of class StaffFrame
public class StaffFrame extends JFrame {

    // The GUI components
    private JPanel labelPanel = new JPanel();
    private JPanel textFieldPanel = new JPanel();
    private JPanel buttonPanel = new JPanel();
    // The two buttons on the frame
    private JButton actionButton = new JButton("Edit");
    private JButton cancelButton = new JButton("Cancel");

    // Constructor
    public StaffFrame(String frameTitle) {
        // Set the frame title
        super(frameTitle);

        // Build the GUI
        Container contentPane = getContentPane();
        contentPane.add(labelPanel, BorderLayout.WEST);
        contentPane.add(textFieldPanel, BorderLayout.CENTER);
    }
}
```

```

        contentPane.add(buttonPanel, BorderLayout.SOUTH);
        buttonPanel.add(actionButton);
        buttonPanel.add(cancelButton);
    }

    // Set the labels and text fields of the frame
    public void setEntryForm(JLabel[] labels, JTextField[] textfields) {
        // Add the labels to the label panel
        labelPanel.setLayout(new GridLayout(labels.length, 1));
        for (int i = 0; i < labels.length; i++) {
            labelPanel.add(labels[i]);
        }

        // Add the text fields to the text field panel
        textFieldPanel.setLayout(new GridLayout(textfields.length, 1));
        for (int i = 0; i < textfields.length; i++) {
            textFieldPanel.add(textfields[i]);
        }

        // Set the dialog to its optimal size
        pack();
    }
}

```

### **ClerkFrame.java**

```

// Resolve classes in java.awt and javax.swing packages
import java.awt.*;
import javax.swing.*;

// Definition of class ClerkFrame
public class ClerkFrame extends StaffFrame {
    // GUI Components
    // Labels
    private JLabel nameLabel = new JLabel("Name: ");
    private JLabel basicSalaryLabel = new JLabel("Basic salary: ");
    // An array object maintaining the labels
    private JLabel[] labels = { nameLabel, basicSalaryLabel };
    // Text fields
    private JTextField nameTextField = new JTextField(20);
    private JTextField basicSalaryTextField = new JTextField(10);
    // An array object maintaining the text fields
    private JTextField[] textfields =
        { nameTextField, basicSalaryTextField };

    // Constructor
    public ClerkFrame() {
        // Set the frame title
        super("Clerk information");
        // Set the labels and text fields
        setEntryForm(labels, textfields);
    }
}

```

**SalesPersonFrame.java**

```
// Resolve classes in java.awt and javax.swing packages
import java.awt.*;
import javax.swing.*;

// Definition of class SalesPersonFrame
public class SalesPersonFrame extends StaffFrame {
    // Labels
    private JLabel nameLabel = new JLabel("Name: ");
    private JLabel basicSalaryLabel = new JLabel("Basic salary: ");
    private JLabel salesVolumeLabel = new JLabel("Sales volumn: ");
    private JLabel commissionRateLabel =
        new JLabel("Commission rate (%): ");
    // An array that maintains all JLabel objects
    private JLabel[] labels =
    {
        nameLabel,
        basicSalaryLabel,
        salesVolumeLabel,
        commissionRateLabel };

    // Text fields
    private JTextField nameTextField = new JTextField(20);
    private JTextField basicSalaryTextField = new JTextField(10);
    private JTextField salesVolumeTextField = new JTextField(10);
    private JTextField commissionRateTextField = new JTextField(10);
    // An array that maintains all JTextField objects
    private JTextField[] textfields =
    {
        nameTextField,
        basicSalaryTextField,
        salesVolumeTextField,
        commissionRateTextField };

    // Constructor
    public SalesPersonFrame() {
        // Set the frame title
        super("Salesperson information");

        // Build the GUI for labels and textfields
        setEntryForm(labels, textfields);
    }
}
```

**ManagerFrame.java**

```
// Resolve classes in java.awt and javax.swing packages
import java.awt.*;
import javax.swing.*;

// Definition of class ManagerFrame
public class ManagerFrame extends StaffFrame {
    // GUI components
    // Labels
    private JLabel nameLabel = new JLabel("Name: ");
    private JLabel basicSalaryLabel = new JLabel("Basic salary: ");
    private JLabel allowanceLabel = new JLabel("Allowance: ");
```

```

// An array maintaining the labels
private JLabel[] labels =
    { nameLabel, basicSalaryLabel, allowanceLabel };
// Text fields
private JTextField nameTextField = new JTextField(20);
private JTextField basicSalaryTextField = new JTextField(10);
private JTextField allowanceTextField = new JTextField(10);
// An array maintaining the text fields
private JTextField[] textfields =
    { nameTextField, basicSalaryTextField, allowanceTextField };

// Constructor
public ManagerFrame() {
    // Set the frame title
    super("Manager information");
    // Set the labels and text fields of the frame
    setEntryForm(labels, textfields);
}

```

### **TestStaffFrame.java**

```

// Definition class TestStaffFrame
public class TestStaffFrame {

    // Main executive method
    public static void main(String args[]) {
        // Create the frames and show them on the screen
        new ClerkFrame().show();
        new SalesPersonFrame().show();
        new ManagerFrame().show();
    }
}

```

### **Self-test 9.5**

As the behaviour for showing the text field contents is common to all dialogs, it is preferable to implement event handling in the superclass instead of the subclass dialogs individually. Therefore, the `StaffFrame` class that maintains the frame title, array objects for the `JLabel` objects, and `JTextField` objects is enhanced. Furthermore, the class is modified to implement the `ActionListener` interface and `actionPerformed()` method is defined, so that the object itself can be the event-handler object of the buttons.

When the <Edit> button is clicked, the array objects of `JLabel` objects and `JTextField` objects are iterated to show the required output in the Command Prompt.

**StaffFrame.java**

```
// Resolve classes in java.awt, java.awt.event and javax.swing package
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

// Definition of class StaffFrame
public class StaffFrame extends JFrame implements ActionListener {

    // The GUI components
    private JPanel labelPanel = new JPanel();
    private JPanel textFieldPanel = new JPanel();
    private JPanel buttonPanel = new JPanel();

    // The arrays storing the labels and the text fields
    private JLabel[] labels;
    private JTextField[] textfields;
    private String title;

    // The two buttons on the frame
    private JButton actionButton = new JButton("Edit");
    private JButton cancelButton = new JButton("Cancel");

    // The button clicked
    private JButton buttonClicked;

    // Constructor
    public StaffFrame(String frameTitle) {
        // Set the frame title
        super(frameTitle);

        // Keep the frame title
        title = frameTitle;
        // Build the GUI
        Container contentPane = getContentPane();
        contentPane.add(labelPanel, BorderLayout.WEST);
        contentPane.add(textFieldPanel, BorderLayout.CENTER);
        contentPane.add(buttonPanel, BorderLayout.SOUTH);
        buttonPanel.add(actionButton);
        buttonPanel.add(cancelButton);

        // Register event handlers with the buttons
        actionButton.addActionListener(this);
        cancelButton.addActionListener(this);
    }

    // Set the labels and text fields of the frame
    public void setEntryForm(JLabel[] labels, JTextField[] textfields) {
        // Keep the supplied label and text field array objects
        this.labels = labels;
        this.textfields = textfields;

        // Add the labels to the label panel
        labelPanel.setLayout(new GridLayout(labels.length, 1));
        for (int i = 0; i < labels.length; i++) {
            labelPanel.add(labels[i]);
        }
    }
}
```



```

        // Add the text fields to the text field panel
        textFieldPanel.setLayout(new GridLayout(textfields.length, 1));
        for (int i = 0; i < textfields.length; i++) {
            textFieldPanel.add(textfields[i]);
        }
        // Set the dialog to its optimal size
        pack();
    }

    public void actionPerformed(ActionEvent e) {
        // Get the source of the event
        Object eventSource = e.getSource();

        // Call a suitable method according to the event source
        if (eventSource == actionButton) {
            actionButtonClicked(e);
        } else if (eventSource == cancelButton) {
            cancelButtonClicked(e);
        }
    }

    // Method called if the left button is clicked
    private void actionButtonClicked(ActionEvent e) {
        // Show the heading
        System.out.println(title);
        // Iterate the labels and text fields to show the contents
        for (int i=0; i < labels.length; i++) {
            System.out.println(labels[i].getText() + "[" +
                textfields[i].getText() + "]");
        }
    }

    // Method called if the Cancel button is clicked
    private void cancelButtonClicked(ActionEvent e) {
        hide();
    }
}

```

### Self-test 9.6

- 1 The Java interface `FactorialCalculator` defines the method `findFactorial`, its parameter list and the return type. Therefore, your boss expects you to write a class that implements the `FactorialCalculator` interface.

The reasons behind your being given the Java interface is that you are mandated to write a method with name, parameter list and return type that matches those defined by the `FactorialCalculator`.

Otherwise, the class is still abstract and the compiler will give you a compile-time error. Therefore, provided that you can give your boss a class that implements the `FactorialCalculator` interface and can be compiled with no compilation error, it is certain that the method mandated by the interface must be implemented.

## 2 InteractiveFactorialCalculator.java

```
// Definition of class InteractiveFactorialCalculator
public class InteractiveFactorialCalculator
    implements FactorialCalculator {

    // Find the factorial of the supplied number
    public long findFactorial(int num) {
        long result = 1L;
        while (num > 1) {
            result *= num;
            num--;
        }

        return result;
    }
}
```

## CachedFactorialCalculator.java

```
// Definition of class CachedFactorialCalculator
public class CachedFactorialCalculator
    implements FactorialCalculator {
    private static int LIMIT = 20;
    private static long NO_RESULT = -1L;
    private long[] result = new long[LIMIT + 1];

    // Constructor for finding the factorials and store them in
    // an array object
    public CachedFactorialCalculator() {
        // The based case for 0! = 1
        result[0] = 1L;
        // A loop is used to find the other factorials
        for (int i=1; i <= LIMIT; i++) {
            result[i] = result[i - 1] * i;
        }
    }

    // Find the factorial of the supplied number
    public long findFactorial(int num) {
        // If the supplied number is within the limit, return the
        // factorial from the array, or NO_RESULT otherwise.
        return (num <= LIMIT) ? result[num] : NO_RESULT;
    }
}
```

## RecursiveFactorialCalculator.java

```
// Definition of class RecursiveFactorialCalculator
public class RecursiveFactorialCalculator
    implements FactorialCalculator {

    // Find the factorial of the supplied number by recursion
    public long findFactorial(int num) {
        return (num == 0) ? 1L : findFactorial(num - 1) * num;
    }
}
```

Defining an interface before writing an implementing class enables the programmer to acquire a better abstraction of the class to be written. The programmer is free from any implementation details and can concentrate on the functionalities of the type only. Then, when the implementing class is written, the interface provides the programmer with the blueprint of the class to be written, and all methods (or behaviours) defined by the interface are mandated and must be implemented.

To illustrate the benefits of using Java interfaces, a driver program `TestFactorials` is written that can create an object of either `IterativeFactorialCalculator`, `CachedFactorialCalculator` and `RecursiveFactorialCalculator` class for actual calculations.

### **TestFactorials.java**

```
// Definition of class TestFactorials
public class TestFactorials {

    // Main executive method
    public static void main(String args[]) {
        // Verify the number of program parameters
        if (args.length < 2) {
            // If there is not sufficient program parameters, show the
            // usage of the program
            System.out.println("java: TestFactorials [type] [times]");
            System.out.println("where");
            System.out.println("type=1, use a loop");
            System.out.println("type=2, use an array");
            System.out.println("type=3, use recursion");
            System.out.println(
                "times - the number of factorials (1!-20!) " +
                "to be obtained");
        }
        else {
            // Determine the type and the number of times to be tested
            int type = Integer.parseInt(args[0]);
            int times = Integer.parseInt(args[1]);

            // The program segment that prepare an object that is
            // subclass of the FactorialCalculator interface
            FactorialCalculator calculator = null;
            switch (type) {
                case 1:
                    calculator = new IterativeFactorialCalculator();
                    break;
                case 2:
                    calculator = new CachedFactorialCalculator();
                    break;
                case 3:
                    calculator = new RecursiveFactorialCalculator();
                    break;
                default:
                    System.out.println("Invalid type");
            }
        }
    }
}
```

```
// The program segment that use a FactorialCalculator
// object
if (calculator != null) {
    for (int i=0; i < times; i++) {
        for (int j=0; j <= 20; j++) {
            long result = calculator.findFactorial(j);
        }
    }
}
}
```

The `TestFactorials` program creates an object for factorial calculations according to the first program parameter. Then, the object is requested to perform the calculation repeatedly. You can see that the object is referred in the program segment with a variable of type `FactorialCalculator` and the variable therefore can refer to an object of the three implementing classes. No matter what the actual class of the object is, no statement in the program segment needs modifications.

If you start the program with an extra option to the JVM, `-Xprof`, you can use the `TestFactorials` program to determine the efficiencies of the three ways for factorial calculations. With the following command

```
java -Xprof TestFactorials 1 3000000
```

the program uses an `InteractiveFactorialCalculator` to obtain 3000000 sets of factorials (from  $0!$  to  $20!$ ), and a report will be shown after the program execution. The report indicates that 14.02 seconds are required with the testing machine.

Similarly, by using the following commands

```
java -Xprof TestFactorials 2 3000000
java -Xprof TestFactorials 3 3000000
```

the efficiencies of the other two implementations can be obtained. The times required are 2.43 and 14.21 seconds respectively. Therefore, the `CachedFactorialCalculator` is the fastest implementation of all. The efficiencies of the `IterativeFactorialCalculator` and `RecursiveFactorialCalculator` are roughly the same, though `InteractiveFactorialCalculator` is slightly faster.

The use of Java interface forces the programmers to use encapsulation. The implementation can be changed easily, provided the implementation is implementing the same Java interface.

**Self-test 9.7****RegionSelectionFrame.java**

```

// Resolve classes in java.awt and javax.swing packages
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

// Definition of class RegionSelectionFrame
public class RegionSelectionFrame
    extends JFrame
    implements WindowListener {
    // GUI components
    private JLabel label = new JLabel("Select the region:");
    private ButtonGroup optionGroup = new ButtonGroup();
    private JRadioButton hkButton =
        new JRadioButton("Hong Kong Island", true);
    private JRadioButton klnButton = new JRadioButton("Kowloon");
    private JRadioButton ntButton = new JRadioButton("New Territories");

    // Constructor
    public RegionSelectionFrame() {
        // Set the frame title
        super("Region Selection");

        // Get the content pane of the frame and set a FlowLayout object
        // to be its layout manager
        Container contentPane = getContentPane();
        contentPane.setLayout(new GridLayout(4, 1));

        // Add the GUI components to the frame
        contentPane.add(label);
        contentPane.add(hkButton);
        contentPane.add(klnButton);
        contentPane.add(ntButton);

        // Associate all radio buttons by a ButtonGroup object
        optionGroup.add(hkButton);
        optionGroup.add(klnButton);
        optionGroup.add(ntButton);

        // Register event handler
        addWindowListener(this);

        // Set the default operation when the close button of the frame
        // is clicked
        setDefaultCloseOperation(DO_NOTHING_ON_CLOSE);

        // Set the GUI to its optimal size
        pack();
    }

```

```
// Method called if the Close button of the frame is clicked
public void windowClosing(WindowEvent e) {
    System.out.print("The selection of the user is: ");
    if (hkButton.isSelected()) {
        System.out.println(hkButton.getText());
    } else if (klnButton.isSelected()) {
        System.out.println(klnButton.getText());
    } else if (ntButton.isSelected()) {
        System.out.println(ntButton.getText());
    }

    // Terminate the program explicitly
    System.exit(0);
}

// The methods corresponding to events that do not need handling
public void windowActivated(WindowEvent we) {}
public void windowClosed(WindowEvent we) {}
public void windowDeactivated(WindowEvent we) {}
public void windowDeiconified(WindowEvent we) {}
public void windowIconified(WindowEvent we) {}
public void windowOpened(WindowEvent we) {}
}
```