Interfaces and Polymorphism
Introduction

You use an interface to define a protocol of behaviour that can be implemented by any class anywhere in the class hierarchy. The code reuse is ultimately made possible by polymorphism.

GUI programs are event-driven. In this module you will learn how to write Java programs that can react to user interface events, such as button pushes and mouse clicks. The Java window toolkit (awt) has a very sophisticated mechanism that allows a program to specify the events in which it is interested and which objects to notify when one of these events occurs.

This module is related to textbook Chapter 9 – Interfaces and Polymorphism.

Objectives

- To learn about interfaces
- To be able to convert between class and interface references
- To understand the concept of polymorphism
- To appreciate how interfaces can be used to decouple classes
- To learn how to implement helper classes as inner classes
- To understand how inner classes access variables from the surrounding scope
- To implement event listeners in graphical applications

Interfaces

Like a class, an interface defines methods. Unlike a class, an interface never implements methods; instead, classes implement the methods defined by the interface. A class can implement multiple interfaces. An interface can be implemented by multiple classes.

When a class implements an interface, the class agrees to implement all the methods defined in the interface. You use an interface to define a protocol of behaviour that can be implemented by any class anywhere in the class hierarchy.

Interfaces are useful for the following (think about ActionListener interface):

- Capturing similarities among unrelated classes without artificially forcing a class relationship
- Declaring methods that one or more classes are expected to implement
- Revealing an object's programming interface without revealing its class
- Modeling multiple inheritance, a feature that some object-oriented languages support that allows a class to have more than one superclass
- An interface type is similar to a class, but there are several important differences:
  - All methods in an interface type are abstract: that is, they have a name, parameters, and a return type, but they do not have an implementation.
  - All methods in an interface type are automatically public.
  - An interface type does not have instance fields.
A class uses the *implements* keyword to indicate that a class implements an interface type. When a class implements an interface, all methods defined in the interface should be implemented. Multiple classes can implement the same interface in different ways. For example, suppose *Measurable* is an interface which has a method *getMeasure()* defined, the class *BankAccount* and class *Coin* implement Measurable interface in the different way, the structure will be like

```java
//interface defines methods
public interface Measurable
{
    double getMeasure();
}

//another class implements interface
public class Coin implements Measurable
{
    //other Coin methods
    public double getMeasure() //implment interface method
    {
        //method implementation 2
    }
}

//another class implements interface
public class BankAccount implements Measurable
{
    //other BankAccount methods
    public double getMeasure() //implement interface method
    {
        //method implementation 1
    }
}
```

Polymorphism can be seen in this way (details in next topic):

```java
Measurable x = new BankAccount();
//call implementation 1
x.getMeasure();

Measurable y = new Coin();
//call implementation 2
y.getMeasure();
```

Note: The behaviour (methods) can vary depending on the actual type of an object.

An example you are familiar with is the ActionListener interface. This interface only has one method defined. Different applications can implement this method in many different ways.

**Polymorphism**

Polymorphism means "many forms". In object-oriented programming, polymorphism refers to a programming language's ability to process objects differently depending on their data type or class. More specifically, it is the ability to redefine methods for derived classes. For example, given a base class *Figure*, polymorphism enables the programmer to define different area methods for any number of derived classes, such as *Rectangle*, and *Triangle*. As a result, invoking the area method for these other Figures will return the correct results. Polymorphism is an essential element of object-oriented programming (OOP).
Converting Between Class and Interface Types

You can think an interface as a superclass. Like class inheritance, you can convert from a class type to an interface type provided the class implements the interface.

For example,

If you have an event listener class like this definition:

```java
//ActionListener is interface, MyListener is your class to implement
//ActionListner
public class MyListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        //do something
    }
}
```

The statements

```java
MyListener listener = new MyListner();
ActionListener a = listener;
```

is legal.

But, in another way, you need a cast operation. This is same as class inheritance.

Inner Class and Accessing Surrounding Variables

An inner class is declared inside another class. Inner classes are commonly used for tactical classes that should not be visible elsewhere in a program. You can declare an inner class inside a method or inside an enclosing class. The inner class inside a method means it is not used beyond the scope of this method. If an inner class is defined inside a class but outside of its methods, the inner class is available to all methods of the enclosing class.

In the GUI programming, if you have many different listeners, it is a good approach to put event listener class as an inner class to each component. In this way, you won’t miss any action listener and it is also easy to maintain.

Methods of an inner class can access variables from the surrounding scope. Local variables that are accessed by an inner-class method must be declared as `final`.

Processing Timer Events

A timer generates timer events at fixed intervals. This feature can be used in programming animations and information updating. The Timer class in the `javax.swing` package generates a sequence of events, spaced apart at even time intervals. Same as event handling we learnt before, you put what action to take into an ActionListner.
Events, Event Sources, and Event Listeners

User interface events include key presses, mouse moves, button clicks, menu selections and so on. Every program must indicate which events it needs to receive. It does that by installing event listener objects. An event listener object belongs to a class that is provided by the application programmers. Its methods describe the actions to be taken when an event occurs. To install a listener, you need to know the event source. The event source is the user interface component that generates a particular event. You add an event listener object to the appropriate event source.

The general structure for defining an event driven application is:

1. Define interface components (event sources) to be used for generating events, such as buttons, text fields, check box etc.
2. Add Event Listeners to those event sources.
3. Define actions performed once the events occur (by implementing an actionPerformed method).

If no event listener object is added to an event source, the event source is inactive; If no action is defined, nothing will happen when the event occurs.

Please evaluate file ClickListener.java and ButtonViewer.java programs on page 409 and 410 of the text.

Building Applications with Buttons

Java’s javax.swing package defines various GUI components classes and their methods. In this section, you are required to familiarize with JButton, JLabel and JPanel classes and the process to build GUI applications. Buttons are the most commonly used components in a Graphic User Interface (GUI).

Use a JPanel container to group multiple user interface components together. For some complex GUI’s, you may overlap panels to achieve a certain layout. Finally, all components have to be put into a main panel; the main panel then has to be put into a frame to form a GUI.

For example:

```java
//to create a JButton object
JButton button = new JButton ("Click Me");
//to create a JLabel object
JLabel label = new JLabel ("This is "+ n + " clicks!");
JPanel panel = new JPanel();  //to create a JPanel object
JFrame frame = new JFrame();  //to create a JFrame object
panel.add(button);
panel.add(label);
frame.add(panel);
```

Once the components are defined and settled in the frame, the next step is to add an event listener that to handle events (button click for a JButton component). You can define another class to implement the ActionListener interface like the example on page 409 and
410 in the text book. You place the button action into the actionPerformed method. You can also install event listeners as inner classes to be able to use fields, methods and final variables. The example, InvestmentViewer1.java, is on page 413.

You may find the following program is easier to understand. You can evaluate it to see how it works.

**Reading**
Chapter 9

**Review questions**

Review exercises:

Programming exercises:
Page 434: Exercise P9.16
Lab session

1) Evaluate the follow program TestButton.java

```java
import java.awt.event.*;
import javax.swing.*;
public class TestButton
{
    private static int n = 0;

    public static void main(String[] args)
    {
        JFrame frame = new JFrame();
        JButton button = new JButton("Click Me");
        final JLabel label = new JLabel("This is "+n+" Click!");
        JPanel panel = new JPanel();
        panel.add(button);
        panel.add(label);
        frame.add(panel);

        //inner class to implement ActionListener
        class AddListener implements ActionListener
        {
            public void actionPerformed(ActionEvent e)
            {
                n++;
                label.setText("This is "+n+" Click!");
            }
        }

        //create an object of AddListener
        ActionListener listener = new AddListener();
        button.addActionListener(listener); //add Listener to the button

        frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
        frame.setVisible(true);
    }

    private static final int FRAME_WIDTH=400;
    private static final int FRAME_HEIGHT=100;
}
```
2) Exercise P9.16 (Put actionListener as separate class)

The GUI class `ButtonViewer` in next page need to use the `ClickListener` class as action listeners for button A and button B, add few statements to `ButtonViewer` class to make the button A and button B activated (i.e., add action listeners to the buttons).

```java
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

/**
 * An action listener that prints a message.
 */
public class ClickListener implements ActionListener
{
    public void actionPerformed(ActionEvent event)
    {
        String name = event.getActionCommand();
        System.out.println("Button " + name + " was clicked!");
    }

    private String name;
}
```
import java.awt.event ActionListener;
import javax.swing JButton;
import javax.swing JFrame;
import javax.swing JPanel;

/**
 * This program demonstrates how to install an action listener.
 */
public class ButtonViewer
{
    public static void main(String[] args)
    {
        JFrame frame = new JFrame();
        JPanel panel = new JPanel();

        JButton buttonA = new JButton("A");
        panel.add(buttonA);

        JButton buttonB = new JButton("B");
        panel.add(buttonB);

        frame.add(panel);
        // add Action Listener to buttons here

        frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }

    private static final int FRAME_WIDTH = 100;
    private static final int FRAME_HEIGHT = 70;
}
3) It is a good practice to use inner classes for listeners and separate main method and the frame, as in the following pattern (you should follow this pattern for your assignment).

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

public class TestButton extends JFrame
{
    private static int n = 0;
    private JButton button;
    private final JLabel label;
    private JPanel panel;
    private static final int FRAME_WIDTH=400;
    private static final int FRAME_HEIGHT=100;

    // constructor for GUI, initialize all components here
    public TestButton()
    {
        button = new JButton ("Click Me");
        label = new JLabel("This is "+ n +" Click!");
        panel = new JPanel();
        panel.add(button);
        panel.add(label);
        add(panel);

        //inner class to implement ActionListener
        class AddListener implements ActionListener
        {
            public void actionPerformed(ActionEvent e)
            {
                n++;
                label.setText("This is " +n+" Click!");
            }
        }
        //create an object of AddListener
        ActionListener listener = new AddListener();

        //add Listener to the button
        button.addActionListener(listener);
    }

    // the main method initialize GUI, should be simple
    public static void main(String[] args)
    {
        JFrame frame = new TestButton();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
        frame.setVisible(true);
    }
}
```
4) Based on the following program structure, what will be the output when the program TestMyFigure (next page) is executed (Interface exercise).

```java
public interface MyFigure {
    public double getArea();
    public double getConditions();
}

public class MyRectangle implements MyFigure {
    private double length, width;
    public MyRectangle(double a, double b) {
        length = a;
        width = b;
    }
    public double getArea() {
        return length * width;
    }
    public double getConditions() {
        return 2 * (length + width);
    }
}

public class MyRecTriangle implements MyFigure {
    private double length, height;
    public MyRecTriangle(double a, double b) {
        length = a;
        height = b;
    }
    public double getArea() {
        return (length * height) / 2;
    }
    public double getConditions() {
        double c = Math.sqrt(length * length + height * height);
        return (length + height + c);
    }
}
```
public class TestMyFigure
{
    public static void main(String[] args)
    {
        MyRectangle r1 = new MyRectangle(2,5);
        MyRecTriangle r2 = new MyRecTriangle(3,6);
        MyFigure f1 = new MyRectangle(3.4, 5.6);
        MyFigure f2 = new MyRecTriangle(3,6);
        System.out.println(r1.getArea());
        System.out.println(r1.getCircumstances());
        System.out.println(r2.getArea());
        System.out.println(r2.getCircumstances());
        System.out.println(f1.getArea());
        System.out.println(f1.getCircumstances());
        System.out.println(f2.getArea());
        System.out.println(f2.getCircumstances());
    }
}