Interface and Polymorphism
Introduction

Except for class inheritance, interface is another mechanism for code reusability. 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. This module covers the topics in Chapter 11 of the textbook.

Objectives

- To learn about interface and polymorphism
- To be able to convert between class and interface references
- 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 for timer events

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

// another class implements interface
public class Coin implements Measurable
{
    // other Coin methods
    public double getMeasure() // implement 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):

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).

Polymorphism can be seen from either class inheritance or interface implementation. The following example demonstrates such a concept in class inheritance. Please examine the programs to see how the objects perform in “many forms”.

3
public class Figure
{
    double dim1;
    double dim2;
    Figure(double a, double b)
    {
        dim1 = a;
        dim2 = b;
    }
    public double area()
    {
        System.out.println("Inside area for Figure.");
        return -1;
    };
}

public class Rectangle extends Figure
{
    Rectangle(double a, double b)
    {
        super(a,b);
    }
    public double area()
    {
        System.out.println("Inside area for Rectangle.");
        return dim1 *dim2;
    }
}

public class Triangle extends Figure
{
    Triangle(double a, double b)
    {
        super(a,b);
    }
    public double area()
    {
        System.out.println("Inside area for Triangle.");
        return dim1 *dim2 / 2;
    }
}

public class AreaTest
{
    public static void main(String[] args)
    {
        Rectangle r = new Rectangle (9,5);
        Triangle t = new Triangle (10,8);
        Figure f;
        f = r;
        System.out.println("Area is " + f.area());
        f = t;
        System.out.println("Area is " + f.area());
    }
}
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 ActionListener
public class MyListener implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        //do something
    }
}
```

The statements

```java
    MyListener listener = new MyListener();
    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 out side 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. You can use inner class in your programming assignment 1 if you like. The MenuFrame.java on page 531 is an example.

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 generated 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 ActionListener. Please read and understand TimerTester.java program on page 429.
Reading

Text book:

Chapter 11: Interfaces and Polymorphism (Whole sections required)

Review questions

Review exercises:

Page 436 - 438: Exercise R11.1, R11.3, R11.6, R11.16

Programming exercises:

Page 439: Exercise P11.13

References

Solution to Programming Exercises

P11-13

```java
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JOptionPane;
import javax.swing.Timer;
import java.util.Date;

/**
 * Displays the current time once every second.
 */
public class ExP11_13
{
    public static void main(String[] args)
    {
        class CurrentTime implements ActionListener
        {
            public void actionPerformed(ActionEvent event)
            {
                Date now = new Date();
                System.out.println(now);
            }
        }

        CurrentTime listener = new CurrentTime();

        final int DELAY = 1000; // milliseconds between timer ticks
        Timer t = new Timer(DELAY, listener);
        t.start();

        JOptionPane.showMessageDialog(null, "Quit?");
        System.exit(0);
    }
}
```