1 Interfaces in Java

Last time we studied inheritance, abstract classes, and polymorphic behaviour. Defining base classes, and deriving classes from the base class is not the only way of achieving polymorphic behavior in Java. It is also possible to use *interfaces* to that effect.

Loosely speaking, an `interface` is a collection of methods which define a certain behavior. Interfaces are defined just like `classes`, but the keyword `interface` is used instead of the keyword `class`. The methods defined within an `interface` are always `public` and `abstract`. You can specify these, but it considered bad form. Since all methods of an `interface` are abstract, a method definition in an `interface` can not have a body.

As usual, we shall continue with an example. Let us define an `interface`, which will be an interface for moving geometric constructs on the plane. Since the name is supposed to express some ability or behavior, we will name the interface `Movable`.

```
public interface Movable {
    void move(double x, double y); // Move relative to current position
    void moveTo(Point p); // Move to absolute position
}
```

So, this `interface` defines a group of methods that can be used to move a geometric construct around in a plane. But, how do we make use of this interface? The answer is, by `implementing` it in a class. How do we do that? Let us show it by example, by implementing the `Movable` interface in the `Circle` class.

```
public class Circle extends Shape implements Movable {
    public void move(double x, double y) {
        center = new Point(center.getX() + x, center.getY() + y);
    }

    public void moveTo(Point p) {
        center = p;
    }
}
```

On the class declaration line, by saying `implements Movable` we promise Java that we are going to implement every method in `Movable` in this class. What happens if we say that, and do not live up to our promise? The code will not compile until we implement all the methods, or declare the class `abstract`.

So why didn't we just put these methods in `Shape`? That would not make sense, because movable things are not limited to `Shapes`. Points, lines, line segments, and whatnot, can also be moved.

Also, a class can extend one and only one class, but it can implement many interfaces. (You just put the interface names, separated by commas, after the `implements` keyword.)

You can declare variables of the `interface`, just like classes. They can refer to any object of a class that implements the interface. So, `interfaces` work as if they were superclasses of the classes that implemented them. This is as far as we are going to investigate `interfaces` right now; this is sufficient for the purposes of learning about AWT events.

2 AWT Events

The AWT provides facilities for user programs that enable the processing of “event”s. An event is some activity from the user, and it comes in many forms; keyboard events, mouse events, mouse movement events, window events, and some more. Each event produces an object of appropriate type, and passes it to the methods that are registered to process that sort of event. The “registration” is done by implementing the relevant interface for the event group in question. As usual, it is best to start right ahead with an example rather than trying to define things in the abstract. The first problem we will attack is the one that has been haunting us for a while – closing the `Frames` that we open.

3 Window Events

Window events are events related to a window (remember that a `Frame` *is* a `Window`). The so-called “window events” are events such as the window being opened, attempted to be closed, being iconified, uniconified and the like. When one of these events happen, a `WindowEvent` object is created, and it passed to the relevant method of the `WindowListener` interface.

Here are the two steps for processing a window event:

1. Implement the `WindowListener` interface in a class. This can be the same class that is going to produce the events.
2. Register that class as a listener for window events of a `Window` object using the `addWindowListener()` method of that object.

Below is a very simple example that implements a frame that can actually be closed:

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

public class ClosableFrame extends Frame implements WindowListener {

    public ClosableFrame() {
        addWindowListener(this);
        setSize(200, 200);
        setLocation(300, 200);
    }

    public void windowDeactivated(WindowEvent e) {
    }
    public void windowClosed(WindowEvent e) {
    }
    public void windowDeiconified(WindowEvent e) {
    }
    public void windowOpened(WindowEvent e) {
    }
    public void windowIconified(WindowEvent e) {
    }
```

```
    public void windowClosing(WindowEvent e) {
        System.exit(0);
    }
    public void windowActivated(WindowEvent e) {
    }

    public static void main(String[] args) {
        ClosableFrame f = new ClosableFrame();
        f.setVisible(true);
    }

}
```

Both the `WindowListener` interface and the `WindowEvent` classes reside in the `java.awt.event` package. Note that the above class implements the `WindowListener` interface, and as such, *must* implement all seven methods that are defined in that interface. Note that in this implementation, the bodies of six of those methods are empty since we do not want to do anything special when those events happen. However, in the case of `windowClosing`, which is called when the user attempts to close the window, we have one new statement we have not used before: `System.exit(0)`. This method, being a static method of `System`, tells the current program to end the execution of all threads, and exit. This is exactly what we want in this case. The argument is an integer which is supposed to inform the system whether the execution was successful or not – a value of zero indicates success, while anything else indicates an error.

Now to the details of the interface and event object:

3.1 The `WindowListener` Interface

The Java documentation gives the following information on the interface `WindowListener`.

```
void windowActivated(WindowEvent e)

Invoked when the window is set to be the user's active window, which
means the window (or one of its subcomponents) will receive keyboard
events.

void windowClosed(WindowEvent e)

Invoked when a window has been closed as the result of calling dispose
on the window.

void windowClosing(WindowEvent e)

Invoked when the user attempts to close the window from the window's
system menu.

void windowDeactivated(WindowEvent e)

Invoked when a window is no longer the user's active window, which
```

```
means that keyboard events will no longer be delivered to the window
or its subcomponents.

void windowDeiconified(WindowEvent e)

Invoked when a window is changed from a minimized to a normal state.

void windowIconified(WindowEvent e)

Invoked when a window is changed from a normal to a minimized state.

void windowOpened(WindowEvent e)

Invoked the first time a window is made visible.
```

3.2 The **`WindowEvent`** Class

An object of type `WindowEvent` is passed to every method in the `WindowListener` interface when a window event occurs. This object carries some information about the event that occured. The `WindowEvent` class has two methods:

```
public Window getWindow()

Returns the originator of the event.

Returns:
the Window object that originated the event

public String paramString()

Returns a parameter string identifying this event. This method is
useful for event-logging and for debugging.

Overrides:
paramString in class ComponentEvent

Returns:
a string identifying the event and its attributes
```

Note that since `WindowEvent` is four steps down in the derivation ladder, it also has a few methods it inherits from the parent class tree. These are examined in the sections below.

4 Action Events

Action events are one of the most important event types in AWT. They are the events that are fired when a user interface component does what it is designed to do. The relevant interface in this case is `ActionListener`. The method for registering to receive action events is `addActionListener()`. The classes that have the addActionListener() method, and thus do generate ActionEvents are as follows:

- Button: An event is generated when the button is pressed.
- List: An event is generated when a list item is double-clicked.
- TextField: An event is generated when **Return** is pressed.

Now to the details of the interface and event object:

4.1 The ActionListener Interface

This interface is a rather simple one – it has only one method defined, hence you only need to write one method in order to implement this interface.

```
public void actionPerformed(ActionEvent e)

Invoked when an action occurs.
```

4.2 The ActionEvent Class

An object of class ActionEvent is passed to the actionPerformed() method when an action event occurs. This object carries information about the event that just occured. The class has three methods which can be used to obtain information about the event:

```
public String getActionCommand()

Returns the command string associated with this action. This string
allows a "modal" component to specify one of several commands,
depending on its state. For example, a single button might toggle
between "show details" and "hide details". The source object and the
event would be the same in each case, but the command string would
identify the intended action.

Returns:
the string identifying the command for this event

public int getModifiers()

Returns the modifier keys held down during this action event.

Returns:
the integer sum of the modifier constants

public String paramString()

Returns a parameter string identifying this action event. This method
is useful for event-logging and for debugging.
```

```
Overrides:
paramString in class AWTEvent

Returns:
a string identifying the event and its associated command
```

5 More about Events

The class hierarchy tree of `ActionEvent` is as follows:

```
java.lang.Object
  |
  +--java.util.EventObject
        |
        +--java.awt.AWTEvent
              |
              +--java.awt.event.ActionEvent
```

And, the class hierarchy tree of `WindowEvent` is as follows:

```
java.lang.Object
  |
  +--java.util.EventObject
        |
        +--java.awt.AWTEvent
              |
              +--java.awt.event.ComponentEvent
                    |
                    +--java.awt.event.WindowEvent
```

As such, both events inherit important methods from their parent classes. We need to examine the methods of the parent classes as well to make full use of the event objects.

5.1 The `EventObject` Class

All event objects are derived from this class. It has two public methods:

```
public Object getSource()

The object on which the Event initially occurred.
Returns:
The object on which the Event initially occurred.

public String toString()

Returns a String representation of this EventObject.
Overrides:
toString in class Object

Returns:
A a String representation of this EventObject.
```

5.2 The `ComponentEvent` Class

Once again, two public methods. The `getComponent()` method is particularly useful.

```
public Component getComponent()

Returns the originator of the event.
Returns:
the Component object that originated the event

public String paramString()

Returns a parameter string identifying this event. This method is
useful for event-logging and for debugging.

Overrides:
paramString in class AWTEvent

Returns:
a string identifying the event and its attributes
```