Java Interfaces and Event Driven Programming

A quick review:

In Java, we can have a class override a method that it would otherwise inherit by creating another method with the exact same name and parameter signature.

The version of the method that is called is determined by the true type. The true type of an object never changes so the overridden version will always be called no matter how the object is typecast.

The typecast only changes the current type.

Java Interfaces

A Java interface is like a class, but it can only contain method stubs (the method header, but no method body) and final fields (and other things that do not involve executable code).

Mostly they just contain method stubs.

To create an interface:

public interface MyInterface {

public void methodStub1(int x, int y);

public int methodStub2();

}

An interface can extend 0 or more interfaces. (You place "extends ..." just like you do with a class, and for multiple interfaces, you separate them with commas.)

To use an interface: a class can implement 0 or more interfaces.

public class MyClass extends Object implements MyInterface {

....

}

If you will implement more than one interface, separate the interface names with commas.

Implementing an interface is -exactly- like extending a class.

So, a class that implements an interface inherits all the methods (or in this case method stubs) of the interface.

A class (that is not abstract) cannot contain method stubs. So, the class must override each of the method stubs from the interface.

The interface is a way to guarantee that a class has certain methods.

Also, MyInterface is above MyClass in the heirarchy. And just like with parent classes, an instance of MyClass is also an instance of MyInterface.

MyClass m = new MyClass();

MyInterface i = m; <- legal. MyInterface is above MyClass in the hierarchy. The current type of i is MyInterface (but the true type is still MyClass). No typecast needed because MyInterface is wider.

m = (MyClass)i; <- legal, but a typecast is needed because we are going down the hierarchy and thus to a narrower type.

Graphical User Interfaces

A graphical user interface is a window or other screen with elements that can be clicked, dragged, typed in, etc.

In Java and on a desktop, we start by creating a JFrame.

To put elements on a JFrame, we have to place them into a panel.

Each panel as a Layout Manager that determines how elements on the panel are organized.

To get the panel for the JFrame, you use

frame.getContentPane()

The default layout manager for a JFrame's content pane is the BorderLayout. The BorderLayout can hold up to 5 different elements, and they go in the locations: North, East, South, West, and Center.

Here is an example:

public class MyButtonFrame extends JFrame {

public MyButtonFrame() {

JButton button = new JButton("Click Me");

this.getContentPane().add(button, "North"); // places the button into the north of the frame

}

}

There are other layout managers available, and in your homework you will use a GridLayout.

Event Driven Programming Paradigm

In the "normal" programming model, a program executes a series of instructions until completion.

In the event driven paradigm, the program waits for an event to happen. For example, it waits until the user clicks a button, and then the program responds to the event.

For this paradigm, we need:

- the program must register with the operating system to receive events

- the operating system needs to know how to inform the program when desired event occurs.

In Java, the JVM handles most of the registering, but we still need to register our objects with the JVM to receive certain events.

For example, we will register our program above to be informed when the button we created was clicked.

The needed method is addActionListener(ActionListener l) of JButton.

public class MyButtonFrame extends JFrame {

public MyButtonFrame() {

JButton button = new JButton("Click Me");

this.getContentPane().add(button, "North"); // places the button into the north of the frame

button.addActionListener(....); // when completed, it registers for the event that this button is clicked

}

}

What is an ActionListener? An interface.

The ActionListener Interface

Java has an ActionListener interface. The addActionListener expects input of type ActionListener, to the object passed to the addActionListener method must be an instance of a class that implements the ActionListener interface.

The ActionListener interface contains a single method stub:

public void actionPerformed(ActionEvent e);

For simplicity, let us have MyButtonFrame implement the ActionListener interface.

public class MyButtonFrame extends JFrame implements ActionListener {

public MyButtonFrame() {

JButton button = new JButton("Click Me");

this.getContentPane().add(button, "North"); // places the button into the north of the frame

button.addActionListener(this); // this object is a MyButtonFrame which implements ActionListener, so this object is an ActionListener

}

public void actionPerformed(ActionEvent e) { // we must override the method stub inherited from ActionListener because our class cannot contain method stubs

System.out.println("The button was clicked");

}

}

The actionPerformed method is what will be called whenever the operating system detects a click of the button.

How does the Java Virtual Machine know that MyButtonFrame has this method? Because the class implements the ActionListener interface.

Java Interfaces and Types

How is JVM using the ActionListener object? Exactly like the interface example at the start of the lecture!

The method addActionListener has its input as type ActionListener, and so inside addActionListener, this object will have current type ActionListener.

addActionListener(this) <- this is being typecast here to ActionListener

Inside addActionListener, the input is stored in a variable, ActionListener listener.

When an action occurs, the JVM calls listener.actionPerformed(some ActionEvent object).

This call is legal because the current type of listener is ActionListener, and ActionListener has an actionPerformed method.

Which method gets called, the true type's overridden version of ActionListener. In our case, that will be the version we wrote in MyButtonFrame.

Multiple Inheritance

Some object-oriented languages such as C++ allow a class to have more than one parent. For example, class C can extend both classes A and B.

This means C will inherit methods from both A and B. What if both A and B have a method m(), and inside class C we call method m()? Whose method is called, A's or B's?

This ambiguity is resolved in C++ by requiring the programmer to indicate the desired parent class when two or more parents have a method of the same name.

In Java, each class can only extend one other class. Thus there is no ambiguity. We can get something like multiple inheritance by allowing a class to implement more than one interface.

There still is no ambiguity because all the interfaces do is state that the class will have a particular method. Only a class can contain a method body (or other executable code).