Government College of Engineering, Jalgaon

Department of Computer Engineering
Experiment No: 12

Subject:CO310U (Application programming Lab) **Sem:**V(Odd)
PRN No:2041024 **Class:**T.Y. B.Tech **Academic Year:**2020-21
Date of Performance: **Date of Completion:**

Aim: Write a program Event handling by anonymous class
Required Software: OpenJDK version "1.8.0_131"
OpenJDK Runtime Environment (build 1.8.0_131-8u131-b11-2ubuntu1.16.04.3-b11)
OpenJDK 64-Bit Server VM (build 25.131-b11, mixed mode)

Java Compiler Version - JAVAC 1.8.0_131

Theory:

Event- In computing, an **event** is an action or occurrence recognized by software that may be handled by the software. Computer events can be generated or triggered by the system, by the user or in other ways. A source of events includes the user, who may interact with the software by way of, for example, keystrokes on the keyboard. Another source is a hardware device such as a timer. Event driven systems are typically used when there is some asynchronous external activity that needs to be handled by a program; for example, a user who presses a button on his mouse. An event driven system typically runs an event loop, that keeps waiting for such activities, e.g. input from devices or internal alarms. When one of these occurs, it collects data about the event and dispatches the event to the *event handler* software that will deal with it.

Delegate Event Model: A common variant in object-oriented programming is the delegate event model, which is provided by some graphic user interfaces. This model is based on three entities:

- a control, which is the event source
- listeners, also called event handlers, that receive the event notification from the source
- interfaces (in the broader meaning of the term) that describe the protocol by which the event is to be communicated.

Furthermore, the model requires that:

- every listener must implement the interface for the event it wants to listen to
- every listener must register with the source to declare its desire to listen to the event

- every time the source generates the event, it communicates it to the registered listeners, following the protocol of the interface.

Event Sources- A source is an object that generates an event. This occurs when the internal state of that object changes in some way. Sources may generate more than one type of event. A source must register listeners in order for the listeners to receive notifications about a specific type of event. Each type of event has its own registration method. Here is the general form:

public void add*Type*Listener(*Type*Listener *el*)

Here, *Type* is the name of the event, and *el* is a reference to the event listener. For example, the method that registers a keyboard event listener is called **addKeyListener( )**. The method that registers a mouse motion listener is called **addMouseMotionListener( )**. When an event occurs, all registered listeners are notified and receive a copy of the event object. This is known as *multicasting* the event. In all cases, notifications are sent only to listeners that register to receive them Some sources may allow only one listener to register. The general form of such a method is this:

public void add*Type*Listener(*Type*Listener *el*) throws java.util.TooManyListenersException

Here, *Type* is the name of the event, and *el* is a reference to the event listener. When such an event occurs, the registered listener is notified. This is known as *unicasting* the event. A source must also provide a method that allows a listener to unregister an interest in a specific type of event. The general form of such a method is this:

public void remove*Type*Listener(*Type*Listener *el*)

Here, *Type* is the name of the event, and *el* is a reference to the event listener. For example, to remove a keyboard listener, you would call **removeKeyListener( )**.
The methods that add or remove listeners are provided by the source that generates events. For example, the **Component** class provides methods to add and remove keyboard and mouse event listeners.

Event Listeners- A listener is an object that is notified when an event occurs. It has two major requirements. First, it must have been registered with one or more sources to receive notifications about specific types of events. Second, it must implement methods to receive and process these notifications. The methods that receive and process events are defined in a set of interfaces found in **java.awt.event**. For example, the **MouseMotionListener** interface defines two methods to receive notifications when the mouse is dragged or moved. Any object may receive and process one or both of these events if it provides an implementation of this interface. Many other listener interfaces are discussed later in this and other chapters.

Event Classes- The classes that represent events are at the core of Java's event handling mechanism. Thus, a discussion of event handling must begin with the event classes. It is important to understand, however, that Java defines several types of events. The most widely used events are those defined by the AWT and those defined by Swing. At the root of the Java event class hierarchy is **EventObject**, which is in **java.util**. It is the superclass for all events. Its one constructor is shown here:

EventObject(Object *src*)

Here, *src* is the object that generates this event.

EventObject contains two methods:
getSource( ) and toString( ).

The **getSource( )** method returns the source of the event. Its general form is shown here:
Object getSource( )

As expected, **toString( )** returns the string equivalent of the event The class **AWTEvent**, defined within the **java.awt** package, is a subclass of **EventObject**.
It is the superclass (either directly or indirectly) of all AWT-based events used by the delegation event model. Its **getID( )** method can be used to determine the type of the event. The signature of this method is shown here: **int getID( )**

At this point, it is important to know only that all of the other classes discussed in this section are subclasses of **AWTEvent**.

To summarize:
• **EventObject** is a superclass of all events.
• **AWTEvent** is a superclass of all AWT events that are handled by the delegation event model. The package **java.awt.event** defines many types of events that are generated by various user interface elements. Commonly used constructors and methods in each class are described in the following sections.

User generated events:

Mouse events- A pointing device can generate a number of software recognizable pointing device gestures. A mouse can generate a number of mouse events, such as mouse move (including direction of move and distance), mouse left/right button up/down and mouse wheel motion, or a combination of these gestures.
Keyboard events- Pressing a key on a keyboard or a combination of keys generates a keyboard event, enabling the program currently running to respond to the introduced data such as which key/s the user pressed.

Touchscreen events- The events generated using a touchscreen are commonly referred to as touch events or gestures.

Device events- Device events include action by or to a device, such as a shake, tilt, rotation, move etc.

Inner class- Inner class is a class that is defined inside another class. Inner class are scoped to the class used to declare them thus they are effectively invisible to the other class in the same package. This lack of visibility to other classes in the package gives the programmer the opportunity to create a set of classes.

Features of inner class-

*. An object of inner class can access the member of outer class.

*. Anonymous inner classes are handy when you want to define callbacks.

*. Inner class can be hidden from other classes in the same package.
Inner classes are the classes that are defined as members of another class. If the inner class is defined at the same level as the enclosing classes' instances/field variables, Inner class can access those instance variables regardless of their access control. This is the same as any method in class that can access the instance variable.

Syntax for inner class

```
Class <outclass>
{ _ _
        Class <inner class>
        {        _
                 _
        }
}
```

Anonymous Inner class- When using a local inner class & we want to make only a single object of this class, you don’t even need to give a class name. such class is called an anonymous inner class. Anonymous inner class can’t have a constructor because the name of the constructor must be the same as the name of the class, and the class has no name. Anonymous inner classes of Java are called *anonymous* because they have no name. They are anonymous and inline. Anonymous classes are essentially inner classes and defined within some other classes.

However, the way anonymous classes are written in Java code may look weird but anonymous inner classes facilitate programmers to declare and instantiate the class at the same time. Another important point to note about anonymous inner classes is that they can be used only once on the place they are coded. In other words, if you want to create only one sub-classed object of a class, then you need not to give the class a name and you can use anonymous inner class in such a case. Anonymous inner classes can be defined not just within a method, but even within an argument to a method. Anonymous inner classes cannot have explicit constructors declared because they have no name to give the constructor.

How to Declare Java Anonymous Inner Classes?

Anonymous inner classes are defined at the same time they are instantiated with new. They are not declared as local classes; rather anonymous inner classes are defined in the new expression itself, as part of a statement. An anonymous inner class declaration expression looks like a constructor invocation, except that there is a class definition contained in a block of code. Before going into further details of anonymous inner classes we must understand that an anonymous inner class is not an independent inner class rather it is a *sub-class* of either a *class* type or an anonymous *implementer* of the specified *interface* type. So, when anonymous inner classes are in picture *polymorphism* must be there. And when polymorphism is there you can only call methods from parent class reference that are defined in the reference variable type. Java's anonymous inner classes being sub-classes or implementers do strictly adhere to the polymorphism rules.

Java Anonymous Inner Class of a Class Type

Let's take an example of seemingly strange looking syntax that defines an anonymous inner class. In the following example code, the Dog reference variable dog refers *not* to an instance of Dog but to an instance of an anonymous inner subclass of Dog.

```
/* AnonymousClassDemo.java */
public class AnonymousClassDemo
{ public static void main(String[] args)
        {
                Dog dog = new Dog() { public
                        void someDog ()
                        {
                                System.out.println("Anonymous Dog");
                        }
                }; // anonymous class body closes here
                        //dog contains an object of anonymous subclass of Dog.
        dog.someDog();
        }
} class
Dog
{ public void someDog()
        {
                System.out.println("Classic Dog");
        }
}
```

In the above piece of code; see the code line Dog dog = new Dog() {, there is a brace at the end of line, not a semicolon. This curly brace opens the class definition and declares a new class that has no name (anonymous class). Now let's enter into the body of newly defined subclass of class Dog and you will see that someDog() is being overridden. This is the crux of defining an anonymous inner class because we want to override one or more methods of the super class on the fly.

Remember, anonymous inner classes are inherited ones, and we always use a superclass reference variable to refer to an anonymous subclass object. And, we can only call methods on an anonymous inner class object that are defined in the superclass. Though, we can introduce new methods in

anonymous inner class, but we cannot access them through the reference variable of superclass because superclass does not know anything about new methods or data members introduced in subclass.
It would be interesting to know that you can also create an anonymous inner class for an interface type. Magically you can also pass anonymous inner class as an argument to a method. We will talk of them in subsequent sections.

Program:

```
/*
Aim: Write a program for Event handling by anonymous class.


import java.awt.*;                    //anonymous inner class
import java.awt.event.*;
//import javax.swing.*; //class
AEvent3 extends JFrame class
AEvent3 extends Frame
{
TextField tf;
AEvent3()
{
tf=new                    TextField();
tf.setBounds(60,50,170,20);
Button b=new Button("click me");
b.setBounds(50,120,80,30);

b.addActionListener(new ActionListener() //anonymous inner class starts here
{
public void actionPerformed(ActionEvent ae){
tf.setText("hello");
}
});  //anonymous inner class ending here
add(b);
add(tf);
setSize(300,300);
setLayout(null);
setVisible(true);
}
public static void main(String args[])
{
AEvent3 f= new AEvent3();
f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); //new
AEvent3();
}
}
```

Conclusion:

Java Anonymous inner class is an inner class without a name and for which only single object is created .Here also we had created inner anonymous class which helps us to make our code more consice and effiecient.

Name and Sign of Teacher