Lab Manual

**OOP**

**Laboratory 11:**

**Statement Purpose:** At the end of this lab, the students should be able to:

* Implement Interface
* Event Handling

**Interface:**

Using the keyword **interface**, you can fully abstract a class interface from its implementation. Interfaces allow you to specify what a class must do, but not how it does it.

Interfaces lack instance variables and their methods are declared without anybody.

Any number of classes can implement an interface. Also, one class can implement any number of interfaces.

By providing the interface keyword, java allows you to fully utilize the **“one interface, multiple methods”** aspect of polymorphism.

**Interface implementation:**

To implement an interface, a class must create the complete set of methods defined by the interface. However, each class is free to determine the details of its own implementation.

**Defining an Interface:**

The general form of the interface is shown below.

access interface name{

return-type method-name1(parameter-list);

return-type method-name2(parameter-list);

type final-varname1=value;

}

**Implementing Interfaces:**

The general form of the class that includes the implements clause looks like this:

access class classname [extends superclass]

[implements interface [,interface..]]{

//class body

}

**Example:**

An example of interface definition is shown below. It declares a simple interface which contains one method called calculate() that takes a single integer parameter.

Project(Lab11\_2k18) -> New -> interface

//defining an interface

**public** **interface** Cal\_Interface {

**void** calculate(**int** param);

}

Project(Lab11\_2k18) -> New -> class

**public** **class** example1 {

**public** **static** **void** main(String[] args) {

//calling method via interface reference variable

Cal\_Interface i\_var=**new** one\_interface();

i\_var.calculate(4);

one\_interface obj = **new** one\_interface();

obj.square\_root(4);

i\_var=**new** two\_interface();

i\_var.calculate(4);

}

}

//implementing interfaces

**class** one\_interface **implements** Cal\_Interface{

**public** **void** calculate(**int** a) //Make this method public explicitly

{

System.*out*.println("First version of calculate");

System.*out*.println("Square of the given number is = " +(Math.*pow*(a, 2)));

}

**void** square\_root(**int** a) //Make this method public explicitly

{

System.*out*.println("\n\*\*\*Another method of class implementing Cal\_Interface\*\*\*");

System.*out*.println("square root of a given number");

System.*out*.println("Square root of " +a+ " is "+Math.*sqrt*(a));

}

}

//Another class implementing interface in a different way

**class** two\_interface **implements** Cal\_Interface{

**public** **void** calculate(**int** a) //Make this method public explicitly

{

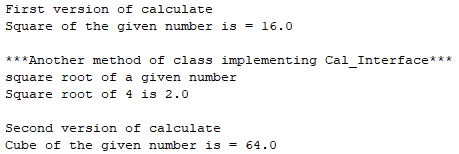
System.*out*.println("\nSecond version of calculate");

System.*out*.println("Cube of the given number is = " +(Math.*pow*(a, 3)));

}

}

**Output:**



**A class implementing two interfaces:**

**public** **interface**  one\_1 { **int** *x*=12;

}

**public** **interface** two\_2 { **int** *y*=10;

**void** display();

}

**public** **class** Demo\_1\_2 **implements** one\_1,two\_2{

**public** **void** display(){

System.*out*.println("X =" +*x*);

System.*out*.println("Y =" +*y*);

System.*out*.println("X+Y =" +(*x*+*y*));

}

}

**public** **class** c\_one\_two {

**public** **static** **void** main(String args[])

{

Demo\_1\_2 d=**new** Demo\_1\_2(); d.display();

}

}

**Output:**



**Interfaces (Verifying Access Specifiers):**

**Aim: To write a Java program to show that the variables in an interface are implicitly static and final and methods are automatically public.**

**public** **interface** one {

**int** *x*=12;

**void** display();

}

**public** **class** COne **implements** one{

**public** **void** display()

{

System.*out*.println("X =" +*x*+ "\n Add 10 to X");

*x*=*x*+10;

System.*out*.println("X =" +*x*);

}

}

**public** **class** Demo\_one\_COne {

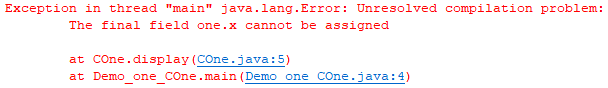
**public** **static** **void** main(String[] args) {

COne o=**new** COne();

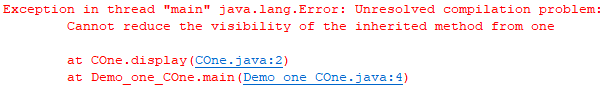
o.display();

}

}



When you remove public access modifier from method display then it will give the following error:



In the above program, the value of the variable x is not defined as final and the display method is not declared to be public in the interface.

But when we try to change the value of x in the implementing class and assign default access privilege to the display method the syntax error occurs, which shows that the variables in the interface are implicitly static and final and methods are public.

**Interfaces can be extended:**

//One interface can extend another

**public** **interface** A {

**void** meth1();

**void** meth2();

}

//B now includes meth1() and meth2() -- it adds meth3().

**public** **interface** B **extends** A{

**void** meth3();

}

//This class implements all of A and B

**public** **class** C\_A\_B **implements** B{

**public** **void** meth1()

{

System.*out*.println("Implement meth1().");

}

**public** **void** meth2()

{

System.*out*.println("Implement meth2().");

}

**public** **void** meth3()

{

System.*out*.println("Implement meth3().");

}

}

//Main class

**public** **class** C\_ab\_main {

**public** **static** **void** main(String args[])

{

C\_A\_B cab=**new** C\_A\_B(); cab.meth1(); cab.meth2(); cab.meth3();

}

}

**Output:**



**Abstract Window Toolkit:**

Java AWT (Abstract Window Toolkit) is an API to develop Graphical User Interface or window-based applications in java.

The java.awt package contains the core AWT graphics classes:

* GUI Component classes (such as Button, TextField, and Label),
* GUI Container classes (such as Frame, Panel, Dialog and ScrollPane),
* Layout managers (such as FlowLayout, BorderLayout and GridLayout),
* Custom graphics classes (such as Graphics, Color and Font).

The java.awt.event package supports event handling:

* Event classes (such as ActionEvent, MouseEvent, KeyEvent and WindowEvent),
* Event Listener Interfaces (such as ActionListener, MouseListener, KeyListener and WindowListener),

**Panel:** Panel in Java is an AWT component which represents a simple container that can attach other GUI components including other panels. It has no border and title bar.

**Frame:**

Frame is a component that works as the main top-level window of the GUI application. It is created using the Frame class. For any GUI application, the first step is to create a frame. There are two methods to create a frame: by extending the Frame class or by creating an object of Frame class.

According to the above program (Example 3), f is a Frame object. Other GUI components are added to it. Finally, the frame is displayed.  The frame is a resizable and a movable window. It has the title bar. The default visibility of a Frame is hidden. The programmer has to make it visible by using setVisible method and providing the value “true” to it.

**Event Handling:**

Event handling is important part of java programming. Events are mostly generated by user. An event is an object that gives state change in component.

The events are supported by “java.awt.event” Package.

**Event listener:**

This interface will catch the occurrence of the particular event and send event information to event handler.

**Event Handler:**

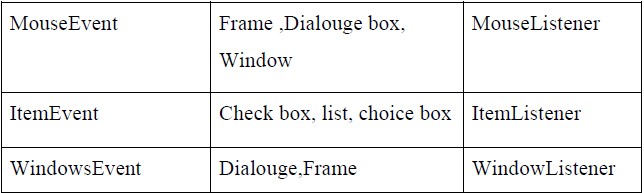
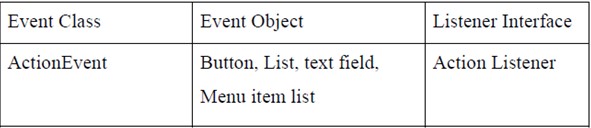
This will handle the particular event. It is a method, which is processed at a particular event.

**Event Sources:**

A source is an object that generated an event. A source is register listener in order for the listener to receive notification about specific event.

**Syntax**: Public void addTypeListner(TypeListner e)

Type is type of event and e is reference to event listener. Commonly used Event classes:



**Handling Mouse Events:**

To handle mouse events must implement the MouseListener and the MouseMotionListener interfaces. It displays the current coordinates of the Mouse in the window. These two interfaces contain methods that receive and process the various types of Mouse events.

**Syntax:**

Void addMouseListener(MouseListener ml)

Void addMouseMotionListener(MouseMotionListener mml)

Here, ml is a reference to the object receiving mouse events, and mml is a reference to the object receiving mouse motion events. In the program the same object is used for both.

**Adding User Controls to Frames:**

1. Make somewhere to display things—usually a Frame or Dialog (for an application),
2. Create some Components, such as buttons, text areas, panels, etc.
3. Add your Components to your display area
4. Arrange, or lay out, your Components
5. Attach Listeners to your Components
6. Interacting with a Component causes an Event to occur.
7. A Listener gets a message when an interesting event occurs and executes some code to deal with it.

In other words, adding user controls is a simple yet relatively long process, to summarize, it involves the following steps:

1. Importing the library in which they are located (java.awt).
2. Declaring user controls variables (or objects).
3. Initializing the controls.
4. Adding the controls in the window.
5. Handling controls events.

**Example: 1**

Here is a simple program which constructs a Frame and an unresponsive Button:

//program to demonstrate the construction of a Container and a Button

**import** java.awt.\*;

**class** Ex1 **extends** Frame

{ // constructor

**public** Ex1(String s)

{

**super**(s); //construct Frame part of Gui

setBackground(Color.*green*);

setLayout(**new** FlowLayout());

Button pushButton = **new** Button("Click");

add(pushButton);

}

}

**public** **class** Awt\_example1 {

**public** **static** **void** main(String[] args) {

Ex1 screen = **new** Ex1("Example 1");

screen.setSize(500,100); //width and height

screen.setVisible(**true**);

}

}

**Output:**

****

**Note:** WindowClosing method is not implemented in it that’s why it will not be closed when you press X.

**Responding to events:**

The next example adds the ability to responds to events, such as a button click.

When an event occurs, when a Button is pressed, an ActionEvent objects is generated.

The ActionListener interface listens for a particular ActionEvent and responds in its actionPerformed method.

The WindowListener interface listens for events associated with Window objects, such as closing a window, and responds in corresponding methods.

**Example :2**

//Program to demonstrate action listeners and event handlers

**import** java.awt.\*;

**import** java.awt.event.\*;

**class** Example2 **extends** Frame **implements** ActionListener, WindowListener

{

//constructor

**public** Example2(String s)

{

**super**(s);

setBackground(Color.*yellow*);

setLayout(**new** FlowLayout());

addWindowListener(**this**); //listen for events on this Window

Button pushButton = **new** Button("click");

add(pushButton);

pushButton.addActionListener(**this**); //listen for Button press

}

//define action for Button click

**public** **void** actionPerformed(ActionEvent event)

{

**if** (event.getActionCommand().equals("click"))

{ System.*out*.print("Clicked");}

}

//define methods in WindowListener interface

**public** **void** windowClosing(WindowEvent event) {

System.*exit*(0);

}

**public** **void** windowClosed(WindowEvent event) {} //do nothing for now

**public** **void** windowDeiconified(WindowEvent event){}

**public** **void** windowIconified(WindowEvent event){}

**public** **void** windowActivated(WindowEvent event){}

**public** **void** windowDeactivated(WindowEvent event){}

**public** **void** windowOpened(WindowEvent event){}

}

**public** **class** Awt\_example2{

**public** **static** **void** main(String[] args) {

Example2 screen = **new** Example2("AWT Example 2");

screen.setSize(500,100);

screen.setVisible(**true**);

} }

**Output:**

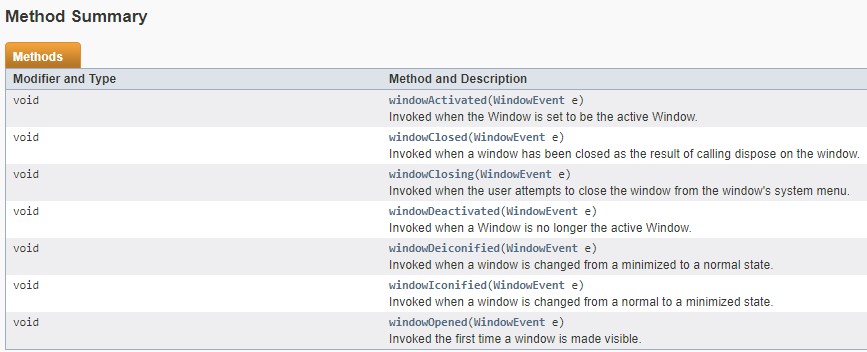


Clicked

The program now has a live button, whose actionPerformed method prints Clicked, as well as a live close window button, which performs System.exit(0).

There are listeners associated with most Components in the AWT.

**Note:** if you implement WindowListener interface, you have to define all of the methods in it. Alternatively, the abstract class WindowAdapter defines null methods for them all, so you can only have to define methods for events you care about, such as windowClosing.



**ActionListener:**

This interface is used to handle the events caused by sources like Buttons, Menu Items, Enter Keys and Double Click of Mouse. When you implements ActionListener Interface following methods need to be override.

public void actionPerformed(ActionEvent ae)

**Example: 3**

**import** java.awt.\*;

**import** java.awt.event.\*;

**public** **class** Example3 **extends** Frame **implements** ActionListener {

Frame f;

Button b1;

TextField t1;

Example3()

{

f=**new** Frame("Button Test");

b1=**new** Button("Submit Form");

t1=**new** TextField(10);

f.add(b1);

f.add(t1);

f.setVisible(**true**);

f.setLayout(**new** FlowLayout());

f.setSize(500,500);

b1.addActionListener(**this**);//Registration of Button with eventhandler

}

**public** **void** actionPerformed(ActionEvent e)//override actionperformed method

{

String str=e.getActionCommand();//method returns string value of button

t1.setText(str);

}

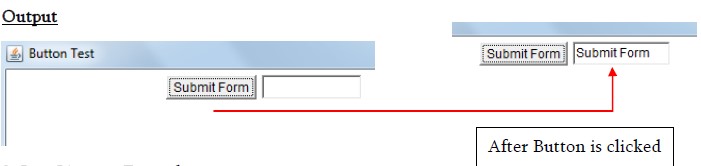
**public** **static** **void** main(String args[])

{

Example3 obj1= **new** Example3();

}

}



**ItemListener:**

This interface is used to handle the events caused by sources like Radio Buttons, Choice and Check Box AWT Controls. The method that needs to be override is as follows: Menu Items, Enter Keys and

public void itemStateChanged(ItemEvent ae)

**Example: 4**

**import** java.awt.\*;

**import** java.awt.event.\*;

**public** **class** Check\_Test **extends** Frame **implements** ItemListener

{

Frame f;

Checkbox c1;

TextField t1;

Check\_Test()

{

f=**new** Frame("Button Test");

c1= **new** Checkbox("CSE", **null**, **false**);

t1=**new** TextField(10);

f.add(c1);

f.add(t1);

f.setVisible(**true**);

f.setLayout(**new** FlowLayout());

f.setSize(500,500);

c1.addItemListener(**this**); //Registration of CheckBox with ItemListener Interface

}

**public** **void** itemStateChanged(ItemEvent ie){ **if**(c1.getState())//Returns true if cehckbox is selected else returns false

{ t1.setText("CSE"); }

}

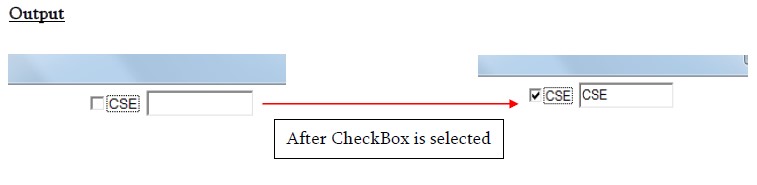
**public** **static** **void** main(String args[])

{

Check\_Test obj1= **new** Check\_Test();

}

}



**Choice:**

The object of Choice class is used to show [popup menu](https://www.javatpoint.com/java-awt-popupmenu) of choices. Choice selected by user is shown on the top of a menu.

Choice c=**new** Choice();

c.setBounds(100,100, 75,75);

c.add("C");

c.add("C++");

 c.add("Java");

 c.add("PHP");

 c.add("Android");

 f.add(c);

**AdjustmentListener:**

This interface is used to handle the events generated from Scrollbar source. This interface has only one method which is

public void adjustmentValueChanged(AdjustmentEvent e)

**Example:**

**import** java.awt.\*;

**import** java.awt.event.\*;

**public** **class** scroll\_test **extends** Frame **implements** AdjustmentListener

{

Frame f;

Scrollbar sb; TextField t1; scroll\_test()

{

f=**new** Frame("Scrollbar Test");

sb= **new** Scrollbar( ); t1=**new** TextField(10);

f.add(sb);

f.add(t1);

f.setVisible(**true**);

f.setLayout(**new** FlowLayout());

f.setSize(500,500);

sb.addAdjustmentListener(**this**);//Registration of Scrollbar with interface

}

**public** **void** adjustmentValueChanged(AdjustmentEvent ie)

{

**int** i= sb.getValue();//Returns ScrollBar value t1.setText(String.*valueOf*(i));

}

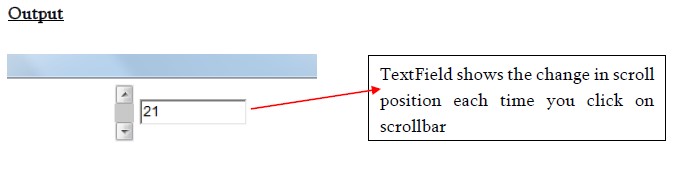
**public** **static** **void** main(String args[])

{

scroll\_test obj1= **new** scroll\_test();

}

}



**KeyListener:**

This interface is used to handle the events generated from the keys of the keyboard. There are three methods present in the KeyListener Interface and all methods needs to be override.

1. public void keyPressesd(KeyEvent ke)
2. public void keyTyped(KeyEvent ke)
3. Public void keyRelaesed(KeyEvent ke)

**Example:**

**import** java.awt.\*; **import** java.awt.event.\*;

**public** **class** key\_event **extends** Frame **implements** KeyListener

{

Frame f;

Label l1;

key\_event()

{

f=**new** Frame("Key Event Test"); l1= **new** Label( );

f.add(l1);

f.setVisible(**true**);

f.setLayout(**new** FlowLayout());

f.setSize(500,500);

f.addKeyListener(**this**);//Registration of KeyListener with Frame

}

|  |
| --- |
| **public** **void** keyTyped(KeyEvent ke){} **public** **void** keyReleased(KeyEvent ke){}  **public** **void** keyPressed(KeyEvent ke)  {  **char** ch= ke.getKeyChar();//Returns the pressed key char  l1.setText(String.*valueOf*(ch));  } |

**public** **static** **void** main(String args[])

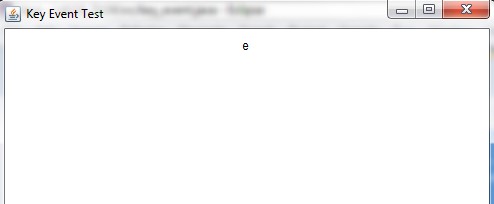
{

key\_event obj1= **new** key\_event();

}

}

**Output:**



**Adapter Classes:**

Adapter classes are used to simplify the process of event handling in java. When we implement any interface all the methods defined in that interface needs to be override in the class, which is not desirable in the case of Event handling. Adapter classes are useful as they provide empty implementation of all methods in an event listener interface. In this case you define a new class to act as event listener by extending one of the adapter classes and implementing only those methods that you want to use in your program.

**Example:**

**import** java.awt.\*;

**import** java.awt.event.\*;

**public** **class** Adapter\_Ex **extends** Frame{

Frame f;

Adapter\_Ex(){

f=**new** Frame("Mouse Event Test");

f.addMouseMotionListener(**new** m1());//Registration of MouseMotionListener with Adapter Class

f.setVisible(**true**);

f.setLayout(**new** FlowLayout());

f.setSize(500,500);

}

**class** m1 **extends** MouseMotionAdapter

{

**public** **void** mouseMoved(MouseEvent me){

**int** x=me.getX();//Returns the X Coordinate Position of mouse

**int** y=me.getY();//Returns the Y Coordinate Position of mouse

System.*out*.println("X Position of mouse is " +x+ "Y Position of mouse is " +y);

}

}

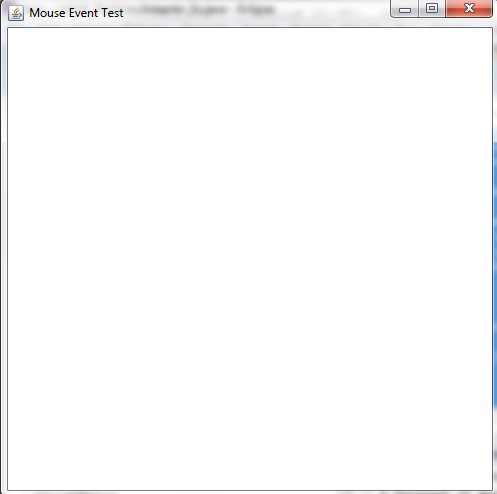
**public** **static** **void** main(String args[])

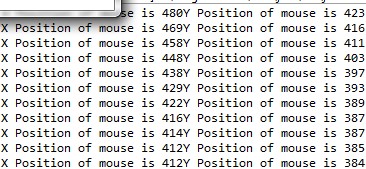
{

Adapter\_Ex obj=**new** Adapter\_Ex();

}

}

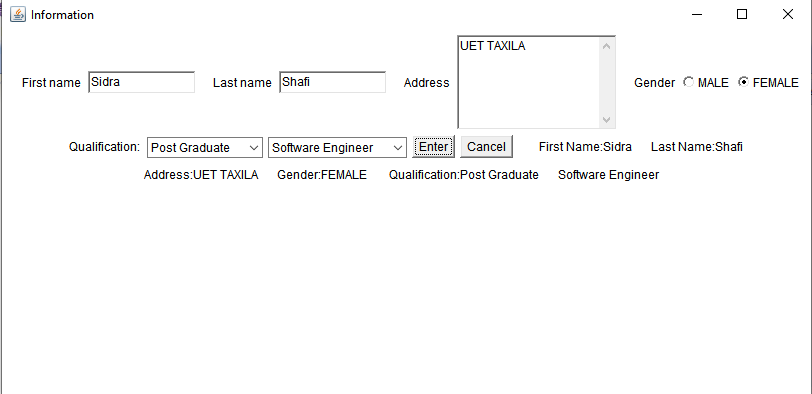




**Lab Task**

**Task 1: Marks: 10**

**Create the following Frame titled “Information” using Abstract Window Toolkit.**

****

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*