Title: Java program using interface.
Aim: Create an applet with three text Fields and four buttons add, subtract, multiply and divide. User will enter two values in the Text Fields. When any button is pressed, the corresponding operation is performed and the result is displayed in the third Text Fields.
Objectives: To learn the use of interface in java.
Theory:

Applet

An applet is a special kind of Java program that is designed to be transmitted over the Internet and automatically executed by a Java-compatible web browser. Furthermore, an applet is downloaded on demand, without further interaction with the user. If the user clicks a link that contains an applet, the applet will be automatically downloaded and run in the browser. Applets are intended to be small programs. They are typically used to display data provided by the server, handle user input, or provide simple functions, such as a loan calculator, that execute locally, rather than on the server. In essence, the applet allows some functionality to be moved from the server to the client. The creation of the applet changed Internet programming because it expanded the universe of objects that can move about freely in cyberspace. In general, there are two very broad categories of objects that are transmitted between the server and the client: passive information and dynamic, active programs. For example, when you read your e-mail, you are viewing passive data. Even when you download a program, the program's code is still only passive data until you execute it. By contrast, the applet is a dynamic, self-executing program. Such a program is an active agent on the client computer, yet it is initiated by the server. As desirable as dynamic, networked programs are, they also present serious problems in the areas of security and portability. Obviously, a program that downloads and executes automatically on the client computer must be prevented from doing harm. It must also be able to run in a variety of different environments and under different operating systems. As you will see, Java solved these problems in an effective and elegant way. Let's look a bit more closely at each.

Applet Fundamentals

Applets are small applications that are accessed on an Internet server, transported over the Internet, automatically installed, and run as part of a web document. After an applet arrives on the client, it has limited access to resources so that it can produce a graphical user interface and run complex computations without introducing the risk of viruses or breaching data integrity.

However, the fundamentals connected to the creation of an applet are presented here, because applets are not structured in the same way as the programs that have been used thus far. As you will see, applets differ from console-based applications in several key areas.

Let's begin with the simple applet shown here:

```
import java.awt.*;
import java.applet.*;
public class SimpleApplet extends Applet {
public void paint(Graphics g) {
g.drawString("A Simple Applet", 20, 20);
}
}
```

This applet begins with two **import** statements. The first imports the Abstract Window Toolkit (AWT) classes. Applets interact with the user (either directly or indirectly) through the AWT, not through the console-based I/O classes. The AWT contains support for a window-based, graphical user interface. As you might expect, the AWT is quite large and sophisticated. Fortunately, this simple applet makes very limited use of the AWT. (Applets can also use Swing to provide the graphical user interface.)

The second **import** statement imports the **applet** package, which contains the class **Applet**. Every applet that you create must be a subclass of **Applet**.
The next line in the program declares the class **SimpleApplet**. This class must be declared as **public**, because it will be accessed by code that is outside the program. Inside **SimpleApplet**, **paint( )** is declared. This method is defined by the AWT and must be overridden by the applet. **paint( )** is called each time that the applet must redisplay its output. This situation can occur for several reasons. For example, the window in which the applet is running can be overwritten by another window and then uncovered. Or, the applet window can be minimized and then restored. **paint( )** is also called when the applet begins execution. Whatever the cause, whenever the applet must redraw its output, **paint( )** is called.
The **paint( )** method has one parameter of type **Graphics**. This parameter contains the graphics context, which describes the graphics environment in which the applet is running. This context is used whenever output to the applet is required. Inside **paint( )** is a call to **drawString( )**, which is a member of the **Graphics** class. This method outputs a string beginning at the specified X,Y location. It has the following general form:
void drawString(String *message*, int *x*, int *y*)
Here, *message* is the string to be output beginning at *x,y*. In a Java window, the upper-left corner is location 0,0. The call to **drawString( )** in the applet causes the message "A Simple Applet" to be displayed beginning at location 20,20.

Notice that the applet does not have a **main( )** method. Unlike Java programs, applets do not begin execution at **main( )**. In fact, most applets don't even have a **main( )** method.

Instead, an applet begins execution when the name of its class is passed to an applet viewer or to a network browser After you enter the source code for **SimpleApplet**, compile in the same way that you have been compiling programs. However, running **SimpleApplet** involves a different process.
In fact, there are two ways in which you can run an applet:

- Executing the applet within a Java-compatible web browser.

- Using an applet viewer, such as the standard tool, **appletviewer**.

An applet viewer executes your applet in a window. This is generally the fastest and easiest way to test your applet.

- ✓ Applets do not need a **main( )** method.

- ✓ Applets must be run under an applet viewer or a Java-compatible browser.

- ✓ User I/O is not accomplished with Java's stream I/O classes. Instead, applets use

the interface provided by the AWT or Swing.

The Applet Class

The **Applet** class is contained in the **java.applet** package. **Applet** contains several methods that give you detailed control over the execution of your applet. In addition, **java.applet** also defines three interfaces: **AppletContext**, **AudioClip**, and **AppletStub**.

Two Types of Applets

It is important to state at the outset that there are two varieties of applets. The first are those based directly on the **Applet** class described in this chapter. These applets use the Abstract Window Toolkit (AWT) to provide the graphic user interface (or use no GUI at all). This style of applet has been available since Java was first created.

The second type of applets are those based on the Swing class **JApplet**. Swing applets use the Swing classes to provide the GUI. Swing offers a richer and often easier-to-use user interface than does the AWT. Thus, Swing-based applets are now the most popular. However, traditional AWT-based applets are still used, especially when only a very simple user interface is required.
Thus, both AWT- and Swing-based applets are valid.

Because **JApplet** inherits **Applet**, all the features of **Applet** are also available in **JApplet**, and most of the information in this chapter applies to both types of applets. Therefore, even if you are interested in only Swing applets, the information in this chapter is still relevant and necessary.
void init( ) :- Called when an applet begins execution. It is the first method called for any applet.
void destroy( ) :- Called by the browser just before an applet is terminated. Your applet will override this method if it needs to perform any cleanup prior to its destruction.

Applet Initialization and Termination

It is important to understand the order in which the various methods shown in the skeletonare called.
When an applet begins, the following methods are called, in this sequence:

1. init( )
2. start( )
3. paint( )

When an applet is terminated, the following sequence of method calls takes place:

1. stop( )
2. destroy( )

init( )

The **init( )** method is the first method to be called. This is where you should initialize variables. This method is called only once during the run time of your applet.

start( )

The **start( )** method is called after **init( )**. It is also called to restart an applet after it has been stopped. Whereas **init( )** is called once—the first time an applet is loaded—**start( )** is called each time an applet's HTML document is displayed onscreen. So, if a user leaves a web page and comes back, the applet resumes execution at **start( )**.

paint( )

The **paint( )** method is called each time your applet's output must be redrawn. This situation can occur for several reasons. For example, the window in which the applet is running may be overwritten by another window and then uncovered. Or the applet window may be minimized and then restored. **paint( )** is also called when the applet begins execution.
Whatever the cause, whenever the applet must redraw its output, **paint( )** is called. The **paint( )** method has one parameter of type **Graphics**. This parameter will contain the graphics context, which describes the graphics environment in which the applet is running. This context is used whenever output to the applet is required.

stop( )

The **stop( )** method is called when a web browser leaves the HTML document containing the applet—when it goes to another page, for example. When **stop( )** is called, the applet is applet is not visible. You can restart them when **start( )** is called if the user returns to the page.

destroy( )

The **destroy( )** method is called when the environment determines that your applet needs to be removed completely from memory. At this point, you should free up any resources the applet may be using. The **stop( )** method is always called before **destroy( )**.

Control Fundamentals

The AWT supports the following types of controls:

- Labels
- Push buttons
- Check boxes
- Choice lists
- Lists
- Scroll bars
- Text editing

These controls are subclasses of **Component**.

Adding and Removing Controls

To include a control in a window, you must add it to the window. To do this, you must first create an instance of the desired control and then add it to a window by calling **add( )**, which is defined by **Container**. The **add( )** method has several forms.

Component add(Component *compObj*)

Here, *compObj* is an instance of the control that you want to add. A reference to *compObj* is returned. Once a control has been added, it will automatically be visible whenever its parent window is displayed.

Sometimes you will want to remove a control from a window when the control is no longer needed. To do this, call **remove( )**. This method is also defined by **Container**. It has this general form:

void remove(Component *obj*)

Here, *obj* is a reference to the control you want to remove. You can remove all controls by calling **removeAll( )**.

Labels

The easiest control to use is a label. A *label* is an object of type **Label**, and it contains a string, which it displays. Labels are passive controls that do not support any interaction with the user. **Label** defines the following constructors:

Label( ) throws HeadlessException

Label(String *str*) throws HeadlessException

Label(String *str*, int *how*) throws HeadlessException

The first version creates a blank label. The second version creates a label that contains the string specified by *str*. This string is left-justified. The third version creates a label that contains the string specified by *str* using the alignment specified by *how*. The value of *how* must be one of these three constants: **Label.LEFT**, **Label.RIGHT**, or **Label.CENTER**.

You can set or change the text in a label by using the **setText( )** method. You can obtain the current label by calling **getText( )**. These methods are shown here:

void setText(String *str*)

String getText( )
For **setText( )**, *str* specifies the new label. For **getText( )**, the current label is returned.
You can set the alignment of the string within the label by calling **setAlignment( )**.
To obtain the current alignment, call **getAlignment( )**. The methods are as follows:
void setAlignment(int *how*)
int getAlignment( )

Buttons

Perhaps the most widely used control is the push button. A *push button* is a component that contains a label and that generates an event when it is pressed. Push buttons are objects of type **Button**. **Button** defines these two constructors:
Button( ) throws HeadlessException
Button(String *str*) throws HeadlessException
The first version creates an empty button. The second creates a button that contains *str* as a label.
After a button has been created, you can set its label by calling **setLabel( )**. You can retrieve its label by calling **getLabel( )**. These methods are as follows:
void setLabel(String *str*)
String getLabel( )
Here, *str* becomes the new label for the button.

- **Handling Buttons**

Each time a button is pressed, an action event is generated. This is sent to any listeners that previously registered an interest in receiving action event notifications from that component. Each listener implements the **ActionListener** interface. That interface defines the **actionPerformed( )** method, which is called when an event occurs. An **ActionEvent** object is supplied as the argument to this method. It contains both a reference to the button that generated the event and a reference to the *action command string* associated with the button. By default, the action command string is the label of the button. Usually, either the button reference or the action command string can be used to identify the button. (You will soon see examples of each approach.)
Here is an example that creates three buttons labeled “Yes”, “No”, and “Undecided”. Each time one is pressed, a message is displayed that reports which button has been pressed. In this version, the action command of the button (which, by default, is its label) is used to determine which button has been pressed. The label is obtained by calling the **getActionCommand( )** method on the **ActionEvent** object passed to **actionPerformed( )**.

```
// Demonstrate Buttons
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="ButtonDemo" width=250 height=150>
</applet>
*/
public class ButtonDemo extends Applet implements ActionListener {
String msg = "";
Button yes, no, maybe;
public void init() {
yes = new Button("Yes");
no = new Button("No");
maybe = new Button("Undecided");
```

```
add(yes);

add(no);
add(maybe);
yes.addActionListener(this);
no.addActionListener(this);
maybe.addActionListener(this);
}
public void actionPerformed(ActionEvent ae) {
String str = ae.getActionCommand();
if(str.equals("Yes")) {
msg = "You pressed Yes.";
}
else if(str.equals("No")) {
msg = "You pressed No.";
}
else {
msg = "You pressed Undecided.";
}
repaint();
}
public void paint(Graphics g) {
g.drawString(msg, 6, 100);
}
}
```

Sample output from the **ButtonDemo** program is shown in Figure 1.

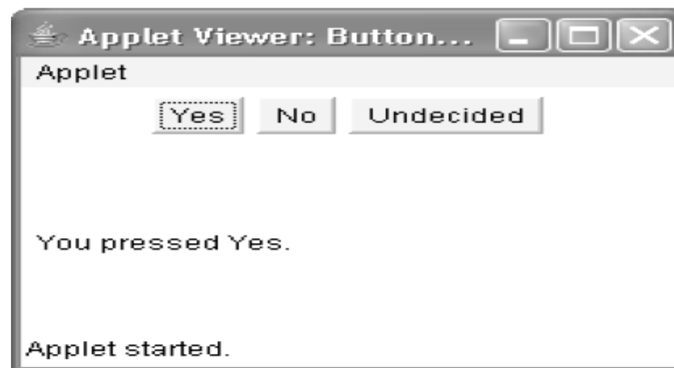

Figure 1 :- Sample output from the ButtonDemo applet

Input:-

Enter the two numbers into the respected text fields.

Output:-

After pressing any button out of four, the desired result will be displayed in third text field.

Conclusion:-

Understand the concept of applet.

References:-

[1] Herbert Schildt, Java: The complete reference, Tata McGraw Hill, 7th Editon.