UNIT- 3 [Around 10+5 = 15 marks]

User Interface Components with Swing

#Concept of AWT:

Java AWT (Abstract Windowing Toolkit) is an API to develop GUI or window-based application in java. Java AWT components are platform-dependent i.e, components are displayed according to the view of operating system. AWT calls Operating Systems subroutine for creating components such as textbox, checkbox, button etc. For example if we are creating a textbox in AWT that means we are actually asking OS to create a textbox for us. This is the reason why AWT components look different on different operating systems. An application build on AWT would look like a windows application when it runs on Windows, but the same application would look like a Mac application when runs on Mac OS.

#Java AWT vs. Java Swing: [Imp]

Java AWT	Java Swing
i) AWT components are platform-dependent.	i) Java Swing components are platform independent.
ii) AWT components are heavyweight.	ii) Swing components are lightweight.
iii) AWT does not support pluggable look and feel.	iii) Swing supports pluggable look and feel.
iv) AWT does not follow MVC.	iv) Swing follows MVC.
v) AWT provides less components than swing.	v) Swing provides more powerful components such as tables, lists, scrollpanes etc.

#Java Applets:

→ Applets are small Java applications which can be accessed on an Internet server, transported over the internet, and can be installed and run automatically as a part of a web document.

→ The applet can create a GUI and it has restricted access to resources so that complicated computations can be carried out without adding the danger of viruses and violating data integrity.

→ Any java applet is a class that extends the class of java.applet.Applet.

→ There is no main() method in an Applet class. The JVM can operate an applet application using either a web-browser plug-in or a distinct runtime environment.

Example of Simple Applet:

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

Benefits of Applets:

→ As it operates on the client side, it requires much less response time.

→ Any browser that has JVM operating in it can operate it.

Applet Life Cycle:

These 4 methods are overridden by most applets. These 4 methods are the lifecycle of the Applet.

init(): The first technique to be called is init(). This is where we initialize the variable. This is called only once during applet runtime.

start(): Method start() is called after init(). This technique is called after it has been stopped to restart an applet.

stop(): Method stop() is called to suspend threads that do not need to operate when the applet is not noticeable.

destroy(): The destroy() method is called if we need to remove our applet from memory entirely.

#Swing Class Hierarchy:

→ Java Swing is a part of Java Foundation Classes (JFC) that is used to create window-based applications. It is built on the top of AWT (Abstract Windowing Toolkit) API and entirely written in java.

→ Unlike AWT, Java Swing provides platform-independent and lightweight components.

→ The javax.swing package provides classes for java swing API such as JButton, JTextField, JTextArea, JMenu, JColorChooser etc.

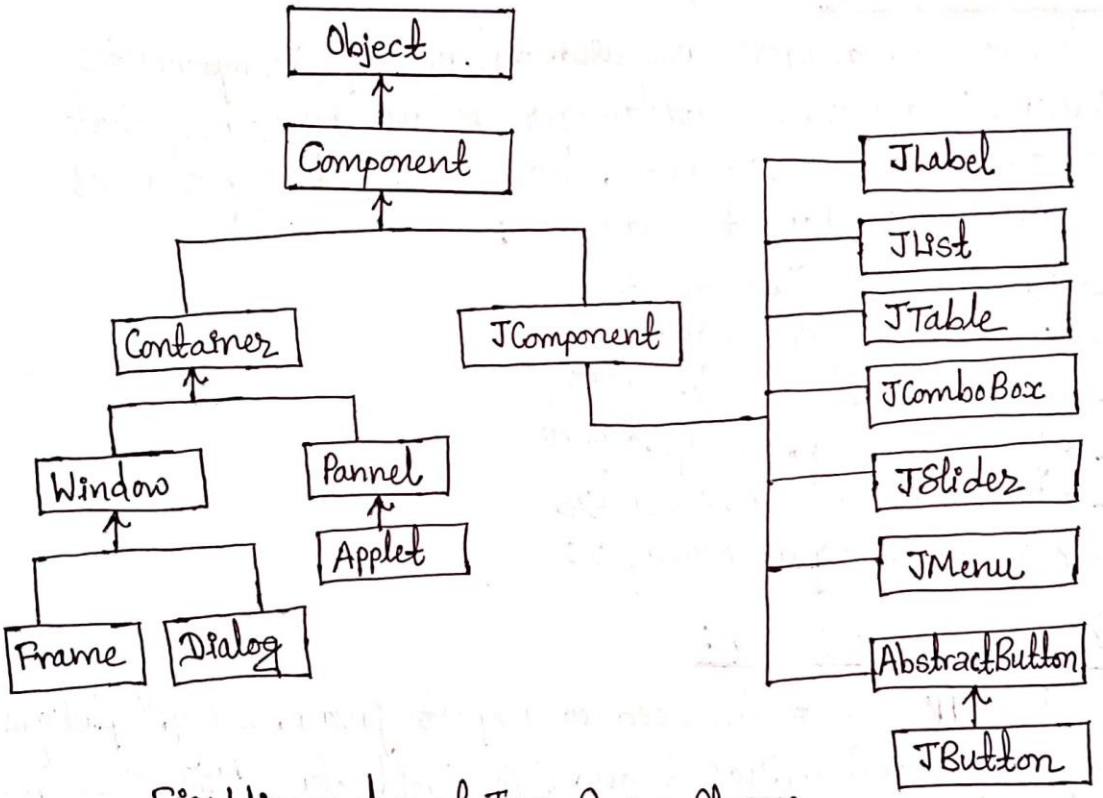


Fig: Hierarchy of Java Swing Classes.

#Components and Containers:

Components: At the top of AWT hierarchy is the Component class. Component is an abstract class that encapsulates all of the attributes of a component. All user interface elements that are displayed on screen and that interact with the user are subclasses of component. It defines over a hundreds of public methods that are responsible for managing events. Component object is responsible for remembering the current foreground and background colors, currently selected text font, and current text alignment.

Containers: The Container class is a subclass of Component. It has additional features that allow other Component objects to be placed within it. Other container objects can also be stored inside of a Container since they are themselves instances of Component. Thus, allowing multilevel containment system. A container is responsible for positioning any component placed on it. It does this through the use of various layout managers. Default layout is present in each container which can be overridden using setLayout method.

→ कुनै एउटा Layout सोधन सक्छ 5 marks को लागि/ 10 marks को लागि दुवै describe any 2 or 3 type सोधन सक्छ

#Layout Management: [Imp]

The layoutManagers are used to arrange components in a particular manner. LayoutManager is an interface that is implemented by all the classes. There are following classes that represents the layout managers:

- → java.awt.BorderLayout
- → java.awt.GridLayout
- → java.awt.GridBagLayout
- → javax.swing.GroupLayout
- → Using No Layout Managers
- → Custom Layout Managers

No Layout (Absolute Positioning):

For No layout we need to import javax.swing.* package library to access JFrame, JLabel, and JTextField class. User Interface of having the LookAndFeelDecorated UI is set to true as below:

```
JFrame.setDefaultLookAndFeelDecorated(true);
```

Then we need to initialize variables in our Main, variable frame as JFrame, label as JLabel, and textField as JTextField.

```
JLabel label = new JLabel("Name:");
JTextField textField = new JTextField("Hello", 15);
```

To set the layout without a layout we will use null keyword in the setLayout method of the frame.

```
frame.getContentPane().setLayout(null);
```

Now we will proceed with the absolute positioning of our components with the use of the setBounds method.

```
label.setBounds(20, 20, 200, 40);
```

And then add method is used to the components.

```
frame.getContentPane().add(label);
frame.getContentPane().add(textField);
```

Lastly, set it's size, visibility to true, and having it's close operation.

```
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.setSize(400,200);
frame.setVisible(true);
```

Flow Layout:

The Java FlowLayout class is used to arrange the components in a line, one after another (in a flow). Fields of FlowLayout class are Left, right, center, leading, trailing etc. and specified as;

```
public static final int CENTER
```

only values are changed here LEFT, RIGHT etc.

FlowLayout(int align, int hgap, int vgap): creates a flow layout with the given alignment, horizontal gap, and vertical gap.

Example:

```
import java.awt.*;
import java.swing.*;
public class FlowLayoutExample {
        JFrame frameObj;
    //Constructor
    FlowLayoutExample(){
            //creating a frame object
              frameObj=new JFrame();
          //creating the button
            JButton b1=new JButton("1");
            JButton b2=new JButton("2");
        //adding buttons to frame
              frameObj.add(b1); frameObj.add(b2);
  //parameter less constructor used so, alignment is center, and hgap and vgap
  // is 5 units.
                frameObj.setLayout(new FlowLayout());
  frame.Obj.setSize(300,300);   frameObj.setVisible(true);
}
public static void main(String args[]){
}}new FlowLayoutExample();
```

for smaller code only two buttons are created and added we can create and add multiple like 6-7 for better view

all examples of layout are almost similar only this line of setLayout may change

BorderLayout:

A border layout places components in upto five areas: top, bottom, left, right, and center. All extra space is placed in the center area. Tool bars that are created using JToolBar must be created with a BorderLayout container, if we want to be able to drag and drop the bars away from their starting positions.

Constructor or Method	Purpose
BorderLayout(int horizontalGap, int verticalGap)	Defines a border layout with specified gaps between components.
setHgap(int)	Sets the horizontal gap between components.
setVgap(int)	Sets the vertical gap between components.

Example:

```
import java.awt.*;
import java.swing.*;
public class Border{
    JFrame f;
    Border(){ f=new JFrame();
              JButton b1=new JButton("Center");
              f.add(b1, BorderLayout.CENTER);
              f.setSize(300,300);
              f.setVisible(true);
            }
    public static void main(String args[]){
        new Border();
    }
}
```

create and add multiple buttons

for detail view notes alternative provided in drive

GridLayout:

GridLayout simply makes a bunch of components equal in size and displays them in the requested number of rows and columns. The constructors used are:

GridLayout(int rows, int cols)
GridLayout(int rows, int cols, int hgap, int vgap)

Example:

```
import java.awt.*;
import java.swing.*;
```

```
public class MyGridLayout{
    JFrame f;
    MyGridLayout(){
        f=new JFrame();
        JButton b1=new JButton("1");
        f.add(b1);
        f.setLayout(new GridLayout(3,3));
        f.setSize(300,300);
        f.setVisible(true);
    }
    public static void main(String args[]){
        new MyGridLayout();
    }
}
```

Create multiple buttons 7-8

add multiple buttons to frame

Similar example Only this line changes setLayout line.

Gridbag Layout:

GridBayhayout is a sophisticated, flexible layout manager. It aligns components to span more than one cell. The rows in the grid can have different heights, and grid columns can have different widths.

```
JPanel pane = new JPanel(new GridBagLayout());
GridBagConstraints c = new GridBagConstraints();
//For each component to be added to this container:
//Create the component...
//Set instance variables in the GridBagConstraints instance...
pane.add(theComponent,c);
```

Group Layout:

GroupLayout is a layout manager that was developed for use by GUI builder tools, but it can also be used manually. GroupLayout works with the horizontal and vertical layouts seperately. The layout is defined for each dimension seperately.

```
GroupLayout layout = new GroupLayout(pannel);
pannel.setLayout(layout);
```

We specify automatic gap insertion:

```
layout.setAutoCreateGaps(true);
layout.setAutoCreateContainerGaps(true);
```

#GUI Controls:

1) Text Input: Text input contains Text Fields, Password Fields, Text areas, Scroll Pane, Label and Labeling Components.

Text Field: A text field is a basic text control that enables the user to type a small amount of text. When the user indicates that text entry is complete (usually by pressing Enter), the text field fires an action event.

Syntax: `textField = new JTextField(20);`

TextArea: To obtain long texts like paragraph which are more than one line of input, we use text area. TextField only takes single line of text while TextArea takes multiple lines of text. It also allows user to edit the text.

Syntax:
```
textArea = new JTextArea(5,20);
JScrollPane = new JScrollPane(textArea);
textArea.setEditable(false);
```
→ for getting scroll option on textArea

→ set true if we want editable

Password Field: A password field provides specialized text fields for password entry. For security reasons, a password field does not show characters that the user types, instead displays different from typed such as an asterisk (*). A password field stores its value as an array of characters, rather than a string.

Syntax:
```
passwordField = new JPasswordField(10);
passwordField.setActionCommand(OK);
passwordField.addActionListener(this);
```

ScrollPane: ScrollPane is used to make scrollable view of a component. When screen size is limited, we use a scroll pane to display a large component.

Syntax: `JScrollPane scrollableTextArea = new JScrollPane(textArea);`

Label: Label is used to display a single line of read only text. The text can be changed by a programmer but a user cannot edit it directly. To create a label, we need to create the object of Label class.

Syntax:
```
l1 = new Label("First");
l1.setBounds(50,100,100,50);
```

2> Choice Components:

Choice components include Check Boxes, Radio Buttons, Borders, Combo Boxes, Sliders.

Check Boxes: Check box is used to turn an option true or false. Clicking on a check box changes its state from true to false or false to true.

Syntax:
```
JCheckbox checkbox = new JCheckBox("Married");
checkbox.setBounds=(100,100,50,50);
```

Radio Button: It is used to choose one option from multiple options. It is widely used in exam systems or quiz. It should be added in ButtonGroup to select one radio button only.

Syntax:
```
JRadioButton r1 =new JRadioButton("Male");
```

Borders: Border component is used to place border in our components.

Syntax:
```
Border bdr=new LineBorder (Color.ORANGE,4,true);
```

Combo Box: The object of Choice class is used to show popup menu of choices. Choice selected by user is shown on the top of menu.

Syntax:
```
String country[]={"Nepal","India","Aus","U.S.A"};
JComboBox cb=new JComboBox(country);
```

Sliders: The Java JSlider class is used to create the slider. By using JSlider, a user can select a value from a specific range.

Syntax:
```
JSlider slider=new JSlider(JSlider.HORIZONTAL,0,50,25);
```

#Menus:

```
//Create the menu bar.
  menuBar =new JMenuBar();
//Build the first menu.
   menu=new JMenu ("A Menu");
   menuBar.add(menu);
//A group of JMenuItems
   menuItem=new JMenuItem ("Both text and icon", new ImageIcon("
                images/myImg.jpg"));
   ....
   menu.addSeperator();
```

//A group of JMenuItems Syntax:

```
menuItem=new JMenuItem("A text-only menu item", KeyEvent.VK_T);
```

Java JMenuItem and JMenu Example:

```
import javax.swing.*;
class MenuExample{
        JMenu menu,submenu;
        JMenuItem i1,i2,i3,i4;
        MenuExample(){
            JFrame f=new JFrame("Menu and MenuItem Example");
            JMenuBar mb=new JMenuBar();
              menu=new JMenu("Menu");
            submenu=new JMenu("Sub Menu");
                i1=new JMenuItem("Item 1");
                i2=new JMenuItem("Item 2");
                i3=new JMenuItem("Item 3");
                i4=new JMenuItem("Item 4");
            submenu.add(i1); submenu.add(i4);
            menu.add(submenu);
            mb.add(menu);
            f.setJMenuBar(mb);
            f.setSize(400,400);
            f.setLayout(null);
            f.setVisible(true);
        }
    public static void main(String args[]){
        new MenuExample();
    }
}
```

⊛ **Icons in Menu Items:** Icons in menu items can be added as:

```
Icon myIcon=new ImageIcon("Resources/myIcon.png");
```

⊛ **Enabling and Disabling Menu Items:**

```
//For disabling
menuItem1.setEnabled(false);
//For enabling
menuItem1.setEnabled(true);
```

⊛. Tooltips:

We can create a tooltip for any JComponent with setToolTipText() method. For example, to add tooltip to PasswordField we do as following:

```
field.setToolTipText("Enter your Password");
```

⊛. CheckBox in MenuItems: & RadioButton in MenuItems:

JCheckBoxMenuItem class represents checkbox which can be included on a menu. A CheckBoxMenuItem can have text or graphic icon or both, associated with it. MenuItem can be selected or deselected. MenuItems can be configured and controlled by actions.

Example:

```
JMenu fileMenu=new JMenu("File");
JCheckBoxMenuItem caseMenuItem= new JCheckBoxMenuItem ("Option 1");
fileMenu.add(caseMenuItem);
```

#Similarly **JRadioButtonMenuItem** class represents RadioButton which can be included on a menu similarly as we did for checkbox.

⊛. Pop-up Menu:

PopupMenu can be dynamically popped up at specific position within a component. It inherits the Menu class.

AWT PopupMenu class declaration

```
public class PopupMenu extends Menu implements MenuContainer,Accessible
```

Example:

```
import java.awt.*;
import java.awt.event.*;
class PopupMenuExample{
    PopupMenuExample(){
        final Frame f=new Frame("PopupMenu Example");
        final PopupMenu popupmenu=new PopupMenu("Edit");
            MenuItem copy=new MenuItem("Copy");
            copy.setActionCommand("Copy");
            MenuItem paste=new MenuItem("Paste");
            copy.setActionCommand("Paste");
            popupmenu.add(copy);
            popupmenu.add(paste);
}
```

```
        f.add(popupmenu);
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
    public static void main(String args[]){
            new PopupMenuExample();
    }
}
```

⊛ Keyboard Mnemonics and Accelerators:

A mnemonic is a key-press that opens a JMenu or selects a MenuItem when the menu is opened. An accelerator is a key-press that selects an option within the menu without it ever being open. The purpose of all this is to let people who really know the program to access functions quickly, and let people that don't use a mouse (some don't) to access the MenuBar.

Example:
```
JMenu menu = new JMenu("Menu"); //Create Menu
menu.setMnemonic('M'); //Set Mnemonic
JMenuItem menuItem = new JMenuItem("Item");
menuItem.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_I,
                                    KeyEvent.SHIFT_MASK));
menu.add(menuItem);
```

⊛ Toolbars:

JToolBar container allows us to group other components, usually buttons with icons in a row or column. JToolBar provides a component which is useful for displaying commonly used actions or controls.

Example:
```
JToolBar toolbar = new JToolBar();
toolbar.setRollover(true);
toolbar.add(new JButton("Edit"));
toolbar.add(new JComboBox(new String[]{"option1", "option2"}));
Container contentPane = myframe.getContentPane();
contentPane.add(toolbar, BorderLayout.NORTH);
contentPane.add
myframe.setSize(450,250);
myframe.setVisible(true);
```

⊛ Option Dialogs: & Creating Dialogs:

The JOptionPane class is used to provide standard dialog boxes such as message dialog box, confirm dialog box and input dialog box. These dialog boxes are used to display information or get input from the user. The JOptionPane class inherits JComponent class.

JOptionPane class declaration:

public class JOptionPane extends JComponent implements Accessible

Example: showMessageDialog():

```
import javax.swing.*;
public class OptionPaneExample{
    JFrame f;
    OptionPaneExample(){
            f=new JFrame();
        JOptionPane.showMessageDialog(f,"Hello");
    }
public static void main(String args[]){
        new OptionPaneExample();
}
}
```

Similarly for showInputDialog():

```
String name=JOptionPane.showInputDialog(f,"Enter Name");
```

Similarly for showConfirmDialog():

```
    f=new JFrame();
    f.addWindowListner(this);
    f.setSize(300,300)
    f.setLayout(null);
    f.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
    f.setVisible(true);
public void windowClosing(WindowEvent e){
    int a=JOptionPane.showConfirmDialog(f,"Are you sure?");
    if(a==JOption.Pane.YES_OPTION){
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

⊛. File Choosers & Color Choosers:

The object of JFileChooser class represents a dialog window from which the user can select file. Similarly The object of JColorChooser class represents a dialog window from ~~the user~~ which the user can select color. They both inherit from JComponent class.

Example: File Choosers:

```
import javax.swing.JFileChooser;
import java.io.File;
public class FileChooserExample(){
    JFileChooser fileChooser=new JFileChooser();
    fileChooser.setCurrentDirectory(new File(System.getProperty("user.home")));
    int result=fileChooser.showOpenDialog(parent);
    if(result==JFileChooser.APPROVE_OPTION){
        //user selects a file
    }
    File selectedFile=fileChooser.getSelectedFile();
    System.out.println("Selected file:"+selectedFile.getAbsolutePath());
}
int public static void main(String args[]){
    new FileChooserExample();
}
```

Example: Color Choosers:

```
import javax.swing.*;
public class ColorChooserExample extends JFrame implements ActionListner{
    JButton b;
    Container c;
    ColorChooserExample(){
        c.getContentPane();
        c.setLayout(new FlowLayout());
        b=new JButton("color");
        b.addActionListner(this);
        c.add(b);
    }
```

```
public void actionPerformed (ActionEvent e){
    Color initialcolor = Color.RED;
    Color color = JColorChooser.showDialog (this, "Select a color", initialcolor);
    c.setBackground (color);
}

public static void main (String args[]){
        ColorChooserExample ch = new ColorChooserExample();
        ch.setSize (400, 400);
        ch.setVisible (true);
        ch.setDefaultCloseOperation (EXIT_ON_CLOSE);
    }
}
```

⊛Internal Frames:

JInternalFrame is a part of Java Swing. JInternalFrame is a container that provides many features of a frame which includes displaying title, opening, closing, resizing, support for menu bar etc.

Example:

```
import java.awt.event.*;
import java.awt.*;
import javax.swing.*;
class solution extends JFrame{
    static JFrame f; //frame
    static label l; //label to display text.
    public static void main(String args[]){
        f= new JFrame ("frame"); //Create new frame
        JInternalFrame in =new JInternalFrame();
        in.setTitle ("Internal frame");

        JButton b=new JButton("button");
        l=new Jlabel ("This is a JInternal Frame");
        JPannel p =new JPannel();
    //add label and button to pannel
        p.add(l); p.add(b);
    in.setVisible(true);
    in.add(p);
    f.add(in);
    f.setSize(300,300);
    f.show();
} }
```

⊛ Advance Swing Components:

1) List: A JList presents the user with a group of items, displayed in one or more columns, to choose from. Lists can have many items, so they are often put in scroll panes.

```
list = new JList(data);
list.setSelectionMode(ListSelectionModel.SINGLE_INTERVAL_SELECTION);
list.setLayoutOrientation(JList.HORIZONTAL_WRAP);
list.setVisibleRowCount(-1);
...
```

2) Trees: With the JTree class, we can display hierarchical data. A JTree object does not actually contain our data, it simply provides a view of data. Like any non-trivial Swing component, the tree gets data by querying its data model.

```
private JTree tree;
...
public TreeDemo(){
...
DefaultMutableTreeNode top = new DefaultMutableTreeNode("Java");
createNodes(top);
tree = new JTree(top);
...
}
```

3) Tables: The JTable class is used to display data in tabular form. It is composed of rows and columns.

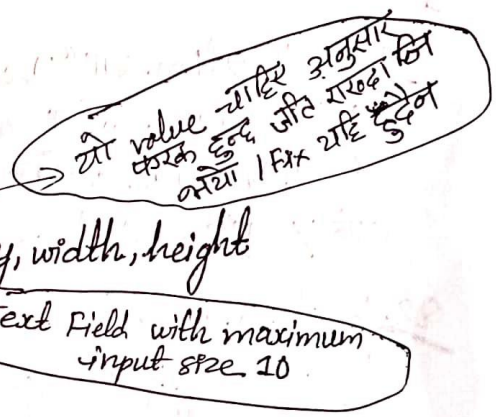
Example:

```
import javax.swing.*;
public class TableExample {
    JFrame f;
    TableExample(){
        f = new JFrame();
        String data[][] = {{"101", "Amit", "67000"},
                           {"102", "Jai", "65000"}};
        String column[] = {"ID", "Name", "SALARY"};
        JTable jt = new JTable(data, column);
        jt.setBounds(30, 40, 200, 300);
        JScrollPane sp = new JScrollPane(jt);
        f.add(sp);
        f.setSize(300, 400);
        f.setVisible(true);
    }
    public static void main(String args[]){
        new TableExample();
    }
}
```

Q. Write a Java program to find sum of two numbers using swing components. Use text fields for input and output. Your program should display the result when the user presses a button. [10 marks] [Imp]

Solution:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

class Addition extends JFrame implements ActionListener{
        JLabel l1,l2,l3;
        JTextField t1,t2,t3;
        JButton b1;
    public Addition(){
        l1 = new JLabel("First Number:");
        l1.setBounds(20,10,100,20); //x,y, width, height
        t1 =new JTextField(10);
        t1.setBounds (120,10,100,20);

        l2=new JLabel("Second Number:");
        l2.setBounds(20,40,100,20);
        t2=new JTextField(10);
        t2.setBounds(120,10,100,20);

        l3 =new JLabel ("Result:");
        l3.setBounds (20,10,100,20);
        t3=new JTextField(10);
        t3.setBounds(120,70,100,120);

        add(l1); add(t1); add(l2); add(t2); add(l3); add(t3);
       b1=new JButton("Sum");
       b1.setBounds(20,70,80,20);
       add(b1);

       b1.addActionListner(this);
       setSize (400,300);
       setLayout (null);
       setVisible (true);
       setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
}
```



```
@Override
public void actionPerformed(ActionEvent e){
        if(e.getSource()==b1){
                int num1=Integer.parseInt(t1.getText());
                int num2=Integer.parseInt(t2.getText());
                int sum=num1+num2;
                t3.setText(String.valueOf(sum));
        }
public static void main(String args[]){
        new Addition();
}
}
```

event button click गरे

Multiply को program आफ्नै लेख्नुस
int product=num1*num2
अनि Addition, Sum लेखेको ठाउमा Multiplication, product लेख्ने अरु सब same राख्ने हो।

Note: Question मा "Use key adapter to handle events" भनेमा यदि same यदि program लेख्ने import गर्दा यो एक line थप्ने "import java.awt.event.KeyAdaptor" र @Override ~~गरेको~~ लाई "class keychecker extends KeyAdaptor{" ले replace गर्ने। [For detail see unitwise question solution in collegenote website:]

Q2. Write a program using swing components to find simple interest. Use text fields for inputs and output. Program should display output if user clicks a button. [Imp]

Solution:

```
//import same all as we did for Q1 before.
class SimpleInterest extends JFrame implements ActionListner{
    //Q1 मा first र second number को लागि गरे जस्तै यसमा
    //चारवटा Label(l1,l2,l3,l4), JTextField(t1,t2,t3,t4), र
    //एउटा button बनाएर add गर्ने same as in Q1.
    //चारवटा textField Principal, Time, Rate, र Simple Interest (i.e, Result) को लागि
                                                                    use गरेको
    :
    :
}
@Override
public void actionPerformed(ActionEvent e){
    if(e.getSource()==b1){
        double P=Double.parseDouble(t1.getText());
        double T=Double.parseDouble(t2.getText());
        double R=Double.parseDouble(t3.getText());
        double SI=(P*T*R)/100;
        t4.setText(String.valueOf(SI));
}
// Main function मा SimpleInterest() call गर्ने as we did for Addition in Q1.
```

फारक यसमा यदि हो int को सट्टा double use हुन्छ

JavaFX

Introduction

JavaFX is a set of Java graphics libraries for creating Java GUI applications, just like Java AWT and Swing.

JavaFX was originally targeted for Rich Interface Application (RIA, introduced in 2002), i.e., GUI webapp delivered thru a browser's plugin (competing with Adobe Flash, Microsoft Silverlight and Java Applets). However, the trend today is to use HTML5/JavaScript-based, instead of plug-in-based framework. Moreover, browsers (such as Firefox) has stopped supporting plug-ins (such as Java Plug-in for Applets).

History

Sun Microsystems created the Java Programming Language and presented JDK 1.0 in 1995/96. To support GUI programming, Java introduced AWT (Abstract Windowing Toolkit) in JDK 1.1 (1997), and Swing in JDK 1.2 (1998). But many developers felt Swing was over-complex and Java on the desktop never really took off as it did on the server.

Sun Microsystems tried several ways to make it easier to create Java GUI applications. One of these was a scripting language called JavaFX Script 1.0 (2008) which allows developers to build much more complex user Interfaces. But JavaFX Script was not Java. It is a totally new language and never really caught on with Java developers.

When Oracle acquired Sun Microsystems, they killed off JavaFX as a scripting language but added its functionality into the Java Language as JavaFX 2.0 (2011). They enhanced it as the new way to develop user interfaces, intended to replace Swing. Starting from JDK 8 (2014), JavaFX was part of JDK (as JavaFX 8).

Oracle will continue to maintain the Swing library but will not enhance it. Swing and JavaFX can be used together. But for writing new Java applications, JavaFX is recommended as it offers a much simpler way to create desktop applications, and you can write more powerful applications with much less code.

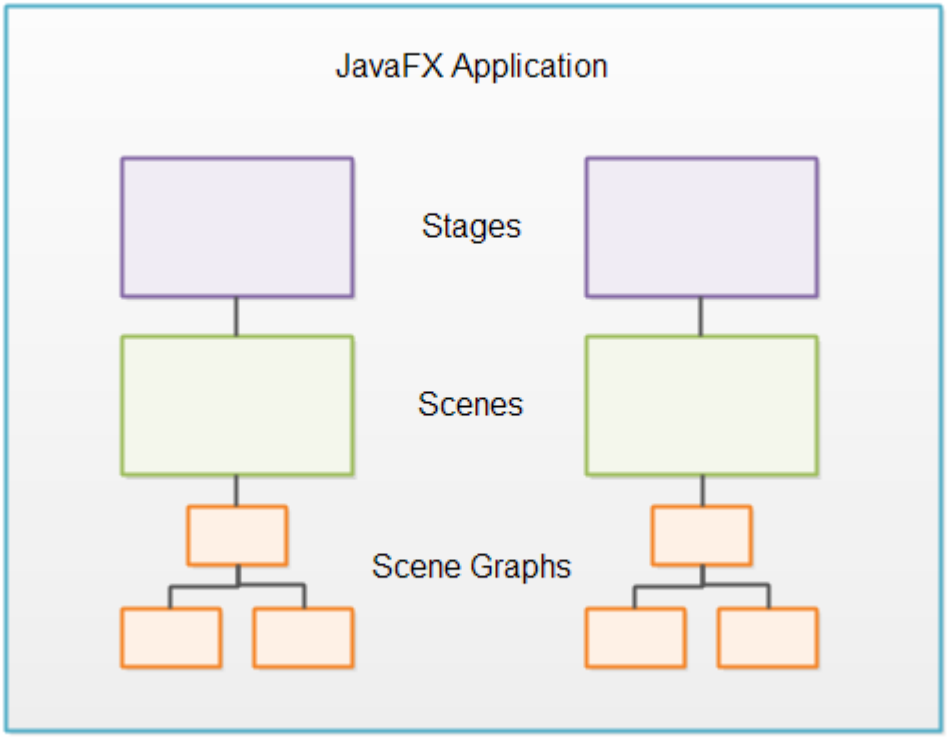
JavaFX Key Features

JavaFX's key features include:

1. From JavaFX 2.0, JavaFX is written in Java (no need to learn a new language). Starting from JDK 8, JavaFX is part of JDK.
2. Support CSS for skinning.
3. Support FXML: a XML-based declarative language to define the structure of the user interface separated from the application code.
4. Swing interoperability: You can use Swing UI in JavaFX application.
5. WebView: for embedding HTML contents.
6. 2D/3D Graphics

7. Media: audio (mp3, wav, aiff), video (flv) and image.
8. Provide a JavaScript engine.

In general, a JavaFX application contains one or more stages which corresponds to windows. Each stage has a scene attached to it. Each scene can have an object graph of controls, layouts etc. attached to it, called the scene graph. These concepts are all explained in more detail later. Here is an illustration of the general structure of a JavaFX application:



Stage

The *stage* is the outer frame for a JavaFX application. The stage typically corresponds to a window. In the early days where JavaFX could run in the browser, the stage could also refer to the area inside the web page that JavaFX had available to draw itself.

Since the deprecation of the Java browser plugin JavaFX is mostly used for desktop applications. Here, JavaFX replaces Swing as the recommended desktop GUI framework. And I must say, that JavaFX looks a whole lot more consistent and feature rich than Swing.

When used in a desktop environment, a JavaFX application can have multiple windows open. Each window has its own stage.

Each stage is represented by a Stage object inside a JavaFX application. A JavaFX application has a primary Stage object which is created for you by the JavaFX runtime. A JavaFX application can create additional Stage objects if it needs additional windows open. For instance, for dialogs, wizards etc.

Scene

To display anything on a stage in a JavaFX application, you need a *scene*. A stage can only show one scene at a time, but it is possible to exchange the scene at runtime. Just like a stage in a theater can be rearranged to show multiple scenes during a play, a stage object in JavaFX can show multiple scenes (one at a time) during the life time of a JavaFX application.

You might wonder why a JavaFX application would ever have more than one scene per stage. Imagine a computer game. A game might have multiple "screens" to show to the user. For instance, an initial menu screen, the main game screen (where the game is played), a game over screen and a high score screen. Each of these screens can be represented by a different scene. When the game needs to change from one screen to the next, it simply attaches the corresponding scene to the Stage object of the JavaFX application.

A scene is represented by a Scene object inside a JavaFX application. A JavaFX application must create all Scene objects it needs.

Scene Graph

All visual components (controls, layouts etc.) must be attached to a scene to be displayed, and that scene must be attached to a stage for the whole scene to be visible. The total object graph of all the controls, layouts etc. attached to a scene is called the *scene graph*.

Nodes

All components attached to the scene graph are called *nodes*. All nodes are subclasses of a JavaFX class called javafx.scene.Node .

There are two types of nodes: Branch nodes and leaf nodes. A branch node is a node that can contain other nodes (child nodes). Branch nodes are also referred to as parent nodes because they can contain child nodes. A leaf node is a node which cannot contain other nodes.

Controls

JavaFX controls are JavaFX components which provide some kind of control functionality inside a JavaFX application. For instance, a button, radio button, table, tree etc.

For a control to be visible it must be attached to the scene graph of some Scene object.

Controls are usually nested inside some JavaFX layout component that manages the layout of controls relative to each other.

JavaFX contains the following controls:

- Accordion
- **Button**
- **CheckBox**
- ChoiceBox
- ColorPicker
- ComboBox

- DatePicker
- **Label**
- ListView
- Menu
- MenuBar
- PasswordField
- ProgressBar
- **RadioButton**
- Slider
- Spinner
- SplitMenuButton
- SplitPane
- TableView
- TabPane
- TextArea
- **TextField**
- TitledPane
- **ToggleButton**
- ToolBar
- TreeTableView
- TreeView

Layouts

JavaFX layouts are components which contains other components inside them. The layout component manages the layout of the components nested inside it. JavaFX layout components are also sometimes called *parent components* because they contain child components, and because layout components are subclasses of the JavaFX class javafx.scene.Parent.

A layout component must be attached to the scene graph of some Scene object to be visible.

JavaFX contains the following layout components:

- Group
- Region
- Pane

- **HBox**
- **VBox**
- FlowPane
- BorderPane
- BorderPane
- StackPane
- TilePane
- GridPane
- AnchorPane
- TextFlow

Nested Layouts

It is possible to nest layout components inside other layout components. This can be useful to achieve a specific layout. For instance, to get horizontal rows of components which are not laid out in a grid, but differently for each row, you can nest multiple HBox layout components inside a VBox component.

Charts

JavaFX comes with a set of built-in ready-to-use chart components, so you don't have to code charts from scratch everytime you need a basic chart. JavaFX contains the following chart components:

- AreaChart
- BarChart
- BubbleChart
- LineChart
- PieChart
- ScatterChart
- StackedAreaChart
- StackedBarChart

2D Graphics

JavaFX contains features that makes it easy to draw 2D graphics on the screen.

3D Graphics

JavaFX contains features that makes it easy to draw 3D graphics on the screen.

Audio

JavaFX contains features that makes it easy to play audio in JavaFX applications. This is typically useful in games or educational applications.

Video

JavaFX contains features that makes it easy to play video in JavaFX applications. This is typically useful in streaming applications, games or educational applications.

WebView

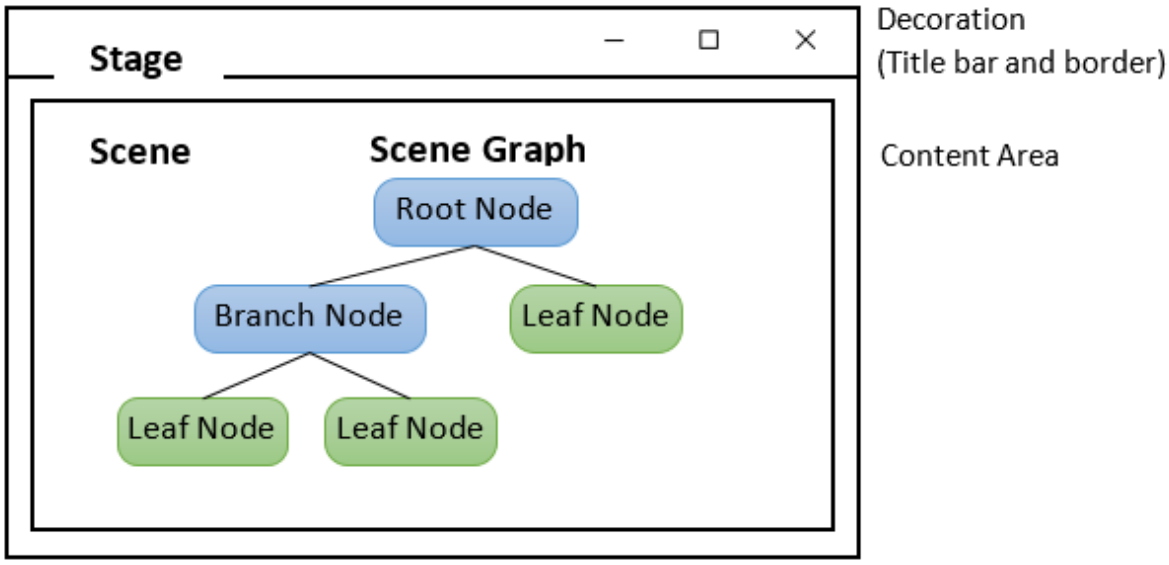
JavaFX contains a WebView component which is capable of showing web pages (HTML5, CSS etc.). The JavaFX WebView component is based on WebKit - the web page rendering engine also used in Chrome and Safari.

The WebView component makes it possible to mix a desktop application with a web application. There are times where that is useful. For instance, if you already have a decent web application, but need some features which can only be provided sensibly with a desktop application - like disk access, communication with other network protocols than HTTP (e.g UDP, IAP etc.) .

JavaFX Application Structure

A JavaFX application (javafx.application.Application) comprises:

1. Stage (javafx.stage.Stage)
2. Scene (javafx.scene.Scene)
3. A hierarchical scene graph of nodes (javafx.scene.Node)



Application and Its Life Cycle

A JavaFX application extends from javafx.application.Application. The JavaFX runtime maintains an Application's life cycle as follows:

1. It constructs an instance of Application.
2. It calls the Application's init() method.
3. It calls the Application's start(javafx.stage.Stage) method, and passes the primary stage as its argument.
4. It waits for the Application to complete (e.g., via Platform.exit(), or closing all the windows).
5. It calls the Application's stop() method.

[TODO] life cycle diagram

The start() is an abstract method, that must be overridden. The init() and stop() has default implementation that does nothing.

If you use System.exit(int) to terminate the application, stop() will not be called.