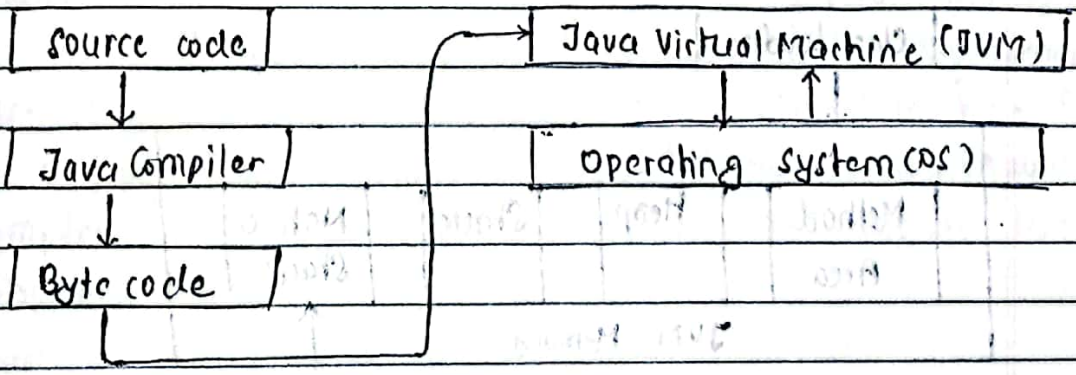
Q NO. 1 Explain the architecture of Java and the significance of JDK, JRE and JVM.

⇒

Java Architecture is a collection of components, i.e. JVM, JRE and JDK. It integrates the process of interpretation and compilation. It defines all the process involved in creating a Java program. Java Architecture explains each and every step of how a program is compiled and executed.

There is a process of compilation and interpretation in Java. Java compiler converts the Java code into byte code. After that the JVM converts the byte code into machine code. The machine code is then executed by the machine.

The following figure represents the Java Architecture in which each step is elaborate graphically,



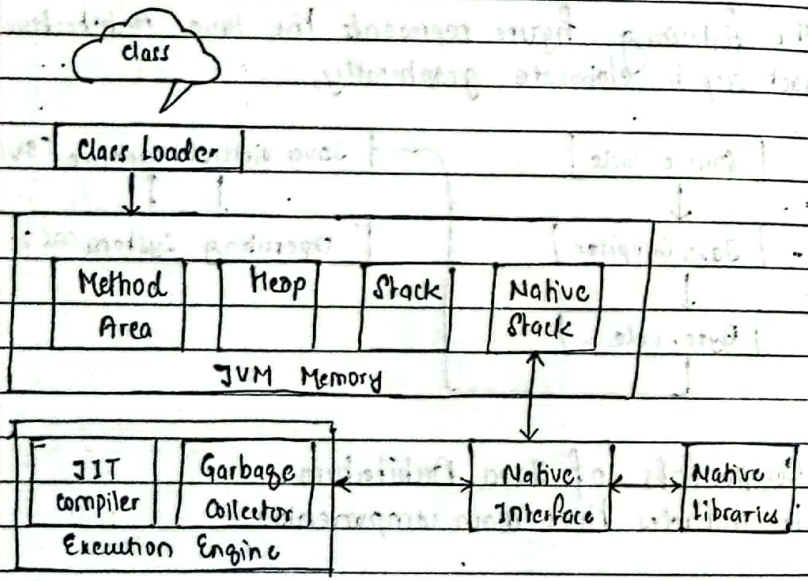

Components of Java Architecture:
It includes three main components:

→ Java Virtual Machine (JVM)
→ Java Runtime Environment (JRE)
→ Java Development Kit (JDK)

(1) Java Virtual Machine

The main feature of Java is WORA. WORA stands for Write Once Run Anywhere. The features states that we can write our code once and use it anywhere or on any operating system. Java Programs are platform independent only because of the Java Virtual Machine. It is a java platform component that gives us an environment to execute java programs. JVM's main task is to convert byte code into machine code.

JVM, first of all, loads the code into memory and verifies it. After that, it executes the code and provides a runtime environment. JVM has it own architecture, which is given below:




(2) Java Runtime Environment (JRE)

It provides an environment in which Java programs are executed. JRE takes our Java code, integrates it with the required libraries, and then starts the JVM to execute it. JRE is the part of the JDK. It is a freely available software distribution which has Java class library, specific tools and a stand-alone JVM. It is the most common environment to run java programs. The JRE loads classes, verify access to memory and retrieves the system resources. JRE acts as a layer on the top of the operating system.

(3) Java Development Kit (JDK)

It is a software development environment used in the development of java applications and applets. Java Development kit holds JRE, a compiler, an interpreter or loader, and several development tools in it. It physically exists. It contains JRE + development tools. The JDK contains a private Java Virtual Machine (JVM) and a few other resources such as Interpreter/loader (java), a compiler (javac), an archiver (jar), etc. to complete the development of Java Application.

Q.NO2. Define classpath.

⇒

In Java, classpath is a parameter or setting used by the Java Virtual Machine (JVM) and the Java compiler to locate user-defined classes, libraries (JAR files) and other resources required during the execution or compilation of a java program.

It specifies the location of .class files or JAR file that the JVM or compiler should reference. It ensures that the required libraries and classes are available at runtime.

Q.NO.3 Explain how Java achieves platform independence.

→

Java is platform-independent because it uses a virtual machine. Java code is compiled into bytecodes. These platform-neutral bytecodes are independent of operating systems and hardware architectures. The JVM, which executes Java programs, allows platform independence. The JVM interprets or JIT compiles bytecodes into machine code for the operating system. Java uses the JVM to compile code once for all hardware architectures and operating systems.

The virtual machine bridges bytecodes to any JVM-compatible system. Developers can write code once and run it on multiple platforms without recompilation. This method provides a consistent programming environment by abstracting the hardware and operating system. Java programs behave uniformly across platforms, making them easy to design and deploy. The Java virtual machine handles hardware-related details, relieving developers from platform-specific compilation, and reducing run-time requirements.

The Java Virtual Machine and bytecode compilation make Java platform independent. The JVM executes platform-neutral bytecodes from the compilation. This potent combination lets Java programs to run across operating systems and hardware architectures, simplifying development.

Q.NO.4 Explain 'Wrapper' Classes.

→

A Wrapper class in Java is a class whose object wraps or contains primitive data types. When we create an object to a wrapper class, it contains a field and in this field, we can store primitive data types. In other words, we can wrap a primitive value into a wrapper class object.

They convert primitive data types into objects. The class in java.util package handles only objects and hence wrapper classes help in this case.

Advantages of Wrapper classes:

1. Collections allowed only object data.
2. On object data, we can call multiple methods compareTo(), equals(), toString().
3. Cloning process only objects
4. Object data allowed null values
5. Serialization can allow only object data.

primitive datatypes and their corresponding wrapper classes:

| primitive data type | wrapper classes |
|---|---|
| → char | Character |
| → byte | Byte |
| → short | Short |
| → int | Integer |
| → long | Long |
| → float | Float |
| → double | Double |
| → boolean | Boolean |

Q.No.5 Define constructor. Write a program to highlight the concept of constructor overloading.

⇒

In Java, a Constructor is a block of codes similar to the method. It is called when an instance of the class is created. At the time of calling the constructor, memory for the object is allocated in the memory. It is a special method that is used to initialize the object. Every time an object is created using the new() keyword, at least one constructor is called.

Syntax:

```
class Class_Name {

    Class_Name(){

    };
}
```

Types:

1. Default constructor
2. Parameterized constructor
3. Copy constructor

Program to illustrate constructor overloading:

```
class Shape{
  int x,y, l,h;

public Shape(int x, int y, int l, int h)
{
  this.x=x;
  this.y=y;
  this.l=l;
  this.h=h;
}
public Shape(int l, int h)
{
  this.l=l;
  this.h=h;
  x=0;
  y=0;
}
public Shape()
{
  l=1;
  h=1;
  x=0;
  y=0;
}
public void getSum(){
  S=x+y+l+h;
System.out.print("sum="+S);
}
}
public class Test {
public static void main (String[] args)
{
  Shape shape_01 = new Shape();
  Shape shape_02 = new Shape(1,2,3,4);
  Shape shape_03 = new Shape(4,6);
  S.O.P("\n Sum of Shape_01:\n");
  Shape_01.getSum();
  S.O.P("\n Sum of Shape_02:\n");
  Shape_02.getSum();
  S.O.P("\n Sum of Shape_03:\n");
```

```
shape.03.getSum();
}
```

O/P:

Sum of shape.01:
2
Sum of shape-02:
10
Sum of shape-03:
10

Q.NO.6 What are the usages of this keyword in Java?

→

The 'this' keyword refers to the current object in a method or constructor.

The most common use of the 'this' keyword is to eliminate the confusion between class attributes and parameters with the same name (because a class attribute is shadowed by a method or constructor parameter).

Some of the usages of this keyword in Java are as follows:

→ Invoke current class constructor
→ Invoke current class method
→ Return the current class object
→ Pass an argument in the method call
→ Pass an argument in the constructor call.
→ Return the current class instance from the method

Q.NO.7 Differentiate betn overloading and overriding with suitable example code.

→

| Method Overloading | Method Overriding |
|---|---|
| 1→ It is a compile-time polymorphism. (static polymorphism) | → It is a run-time polymorphism (dynamic polymorphism) |
| 2→ Two or more functions in the same class with the same name but diffn parameters or signature | → A derived class provides its own implementation of a function that is already defined in the base class. |
| 3→ Not dependent on inheritance | → Dependent on inheritance. |
| 4→ parameters must be different | → parameters must be same. |
| 5→ Return type may or may not be same. | → Return type must be same |
| 6→ Private and final methods can be overloaded | → Private and final methods can't be overridden. |
| 7→ Method overloading helps to increase the readability of the program. | → It is used to grant the specific implementation of the method which is already provided by its parent class or superclass. |

Example of Method overloading:

```
class MethodOverloadingEx {

static int add (int a, int b) {
    return a+b; }
```

```
static int add(int a, int b, int c) {
return a+b+c;
}
public static void main(String [] args)
{
S.O.P("add() with 2 parameters"); S.O.P("add(4,6)");
S.O.P("add() with 3 parameters");
S.O.P(add(4,6,7));
}
}
```

O/P:

add() with 2 parameters
10
add() with 3 parameters
17

Example of Method Overriding:

```
class Animal {
  void eat() {
S.O.P("eat() method of base class");
S.O.P("Animal is eating");
}
}
class Dog extends Animal {
    @Override
    void eat() {
    S.O.P("eat() method of derived class");
    S.O.P("Dog is eating");
}
  void eatAsAnimal() {
  super.eat();
} }
```

```
class MethodOverridingEx {
public static void main(String[]args)
{
  Dog ds = new Dog();
  Animal a1 = new Animal();
  ds.eat();
  a1.eat();
Animal animal = new Dog();
animal.eat();
((Dog) animal).eatAsAnimal();
}
}
```

O/P:

eat() method of derived class
Dog is eating
eat() method of base class
Animal is eating
eat() method of derived class
Dog is eating
eat() method of base class
Animal is eating.

QNO.8 Why is multiple inheritance not possible in Java?

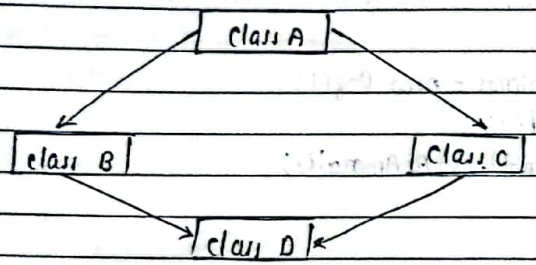
⇒

The major reason behind Java's lack of support for multiple inheritance lies in its design philosophy of simplicity and clarity over complexity. By disallowing Multiple inheritance, Java aims to prevent the ambiguity and complexities that can arise from having multiple parent classes.

The diamond problem is the classic issue that can arise with multiple inheritance. It occurs when the class inherits from the

two classes that have a common ancestor. This situation forms the diamond-shaped inheritance hierarchy. It can lead to the ambiguity in the method resolution.

Diamond Problem in Java:




In order to eliminate this problem and to reduce the complexity, multiple inheritance is not supported in Java.

QNO.9 Using Interfaces, write a sample program to show how multiple inheritance can be achieved.

=>

```
Interface InterfaceA {
    void methodA();
}

Interface InterfaceB {
    void methodB();
}

class MyClass implements InterfaceA, InterfaceB {

public void methodA() {
S.O.P("A");
}

public void methodB() {
S.O.P("B");
}
}

class Main {
public static void main(String[] args)

InterfaceA x = new MyClass();

InterfaceB y = new MyClass();
x.methodA();
y.methodB();
}
}
```

O/P:
A
B

QNO.10 Differentiate betn interface and abstract class:

=>

| Interface | Abstract class. |
|---|---|
| (1) It contains only abstract methods but since Java 8, we have default and static methods also | 1. It may have abstract or non abstract methods. |
| (2) It supports multiple inheritance | 2. It doesn't support multiple inheritance |
| (3) It cannot extends from a class or an abstract class. | 3. It can provide implementation of an interface. |
| (4) It can have final and static members. | 4. It can have final, non-final, static and non-static members. |

| | | |
|---|---|---|
| (5) | All members of an interface have public scope. | (5) It can have public, private, & protected scope |
| (6) | It cannot have a constructor | (6) It may have a constructor |
| (7) | It provides abstraction level of 100%. | (7) It can provide abstraction level of 0 to 100%. |
| (8) | It has 'interface' keyword. | (8) It has 'abstract' keyword. |

QNO.11 Explain Error and Exception. How can exception be handled in Java?

⇒

Errors and exceptions are both types of throwable objects, but they represent different types of problems that can occur during the execution of a program.

Errors:

→ These are problems that mainly occur due to the lack of system resources. It cannot be caught or handled. It indicates a serious problem. It occurs at runtime. These are always unchecked. Errors are represented by the Error class and its subclasses. Some common examples of errors in Java are:

(1) OutOfMemoryError: Thrown when the Java Virtual Machine (JVM) runs out of memory.
(2) StackOverflowError: Thrown when the call stack overflows due to too many method invocations.
(3) NoClassDefFoundError: Thrown when a required class cannot be found.

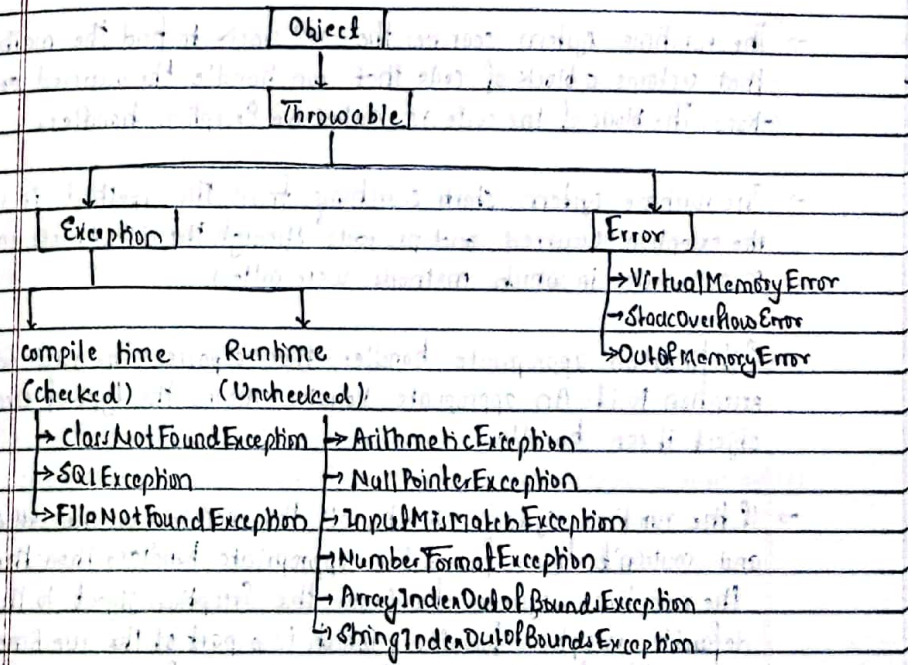
Exceptions:

→ It is an event that occurs during the execution of the program and interrupts the normal flow of program instructions.

These are the errors that occurs at compile time and runtime. It occurs in the code written by the developers. Exceptions are represented by the Exception class and its subclasses. Some common examples of exceptions in Java include:

→ NullPointerException: Throws when a null reference is accessed.
→ IllegalArgumentException: Thrown when an illegal argument is passed to a method.
→ IOException: Thrown when an I/O operations fails.

The table below shows the error, exception and their classes:



## Exception Handling in Java:

Whenever inside a method, if an exception has occurred, the method creates an object known as Exception object and hands it off to the run-time system (JVM). The exception object contains the name and the current state of the program where the exception has occurred. Creating the exception object and handling it in the run-time system is called throwing an exception. There might be a list of the method where an exception occurred. This ordered list of methods is called callstack. Now the following procedure will happen:

→ The run-time system searches the call stack to find the method that contains a block of code that can handle the occurred exceptions. The block of the code is called an Exception handler.

→ The runtime system starts searching from the method in which the exception occurred and proceeds through the call stack in the reverse order in which methods were called.

→ If it finds an appropriate handler, then it passes the occurred exception to it. An appropriate handler means the type of exception object it can handle.

→ If the run-time system searches all the methods on the call stack and couldn't have found the appropriate handler, then the the run-time system handover the Exception object to the default exception handler, which is a part of the run-time system. This handler prints the exception information in the certain format and terminates the program abnormally.

Q.NO. 12 Explain any four common exceptions with an example of possible occurrence for each type.

⇒ Some of the common exceptions in Java are explained below:

1. Arithmetic Exception:

It is thrown when an exceptional condition has occurred in an arithmetic operation.

example:

```
class ArithmeticException_Demo {
public static void main (String [] args)
{
  try { int a=30, b=0;
        int c=a/b; // cannot be divide by zero
        System.out.print ("Result=" +c);
  }
  catch (ArithmeticException e)
    { System.out.println ("Can't divide a number by 0");
    }
}
```

2. NullPointerException:

It is raised when referring to the members of a null object. Null represents nothing.

For example:

```
class NullPointer_Demo {
public static void main (String [] args)
 { try {
    String a = null; //null value
    System.out.println (a.charAt (0));
  }
```

```
catch (NullPointerException e){
  System.out.println ("NullPointerException");
  }
  }
}
```

3. StringIndexOutOfBoundException

It is thrown by String class methods to indicate that an index is either negative or greater than the size of the string.

Example:

```
class StringIndexOutOfBound_Demo
{ public static void main (String [] args) {
  try { String a = "The crisis is coming";
  char c = a.charAt (24);
   System.out.println (c);
  }
   catch (StringIndexOutOfBoundsException e) {
  System.out.println ("StringIndexOutOfBoundsException");
  }
  }
}
```

4. NumberFormatException:

This exception is raised when a method could not convert a string into a numeric format.

For example:

```
class NumberFormat_Demo{
public static void main (String [] args)
{ try {
    int num = Integer.parseInt ("aka");
    System.out.println (num);
  }
  catch (NumberFormatException e) {
   System.out.println ("Number format Exception");
  }
  }
}
```

Q.No.13 Create a sample custom Exception class and write a program showing its use.

⇒

Custom Exception handler to handle arithmetic exception:

```
class ErException extends Exception {

public ErException (String msg) {
    super (msg);
  }
}
 class Div{
   int a,b;
public Div (int a, int b) throws ErException{
try{
   System.out.println (a/b);
}
  catch (ArithmeticException e)
{
throw new ErException ("can't divide by 0");
 }
}
class Test { public static void main (String[] args.) {
{ Div d = new Div (5,2);
 }
}
```

Q.NO.14 Differentiate betn Swing and AWT.

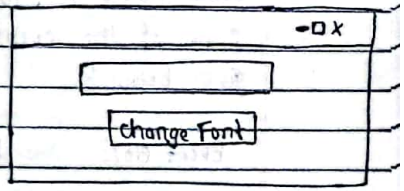
⇒

| AWT | Swing |
|---|---|
| 1. It stands for Abstract Window Toolkit. | 1. It is a part of Java Foundation classes (JFC). |
| 2. It is platform dependent since it uses components of underlying OS & operating system. | 2. It is platform independent & has its own set of components. |
| 3. Its execution speed is slower. | 3. Its execution speed is faster. |
| 4. Components of AWT are heavy weight. | 4. Components of swing are light weight. |
| 5. It doesn't have pluggable looks and feels. | 5. It has pluggable looks & feel. |
| 6. It has limited no. of components compared to swing. | 6. It has extensive number of components. |
| 7. Components of AWT are less customizable. | 7. Its components are highly customizable. |

Q.NO.15 Create an GUI application with a textfield and a button. When the button is pressed the following tasks should be performed:
(a) Change the text field's text into Red.
(b) Change the fontstyle of the text to Arial, Bold and size of 20pts.

⇒
```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class TextFieldFontChanger {

public static void main (String[] args){
JFrame frame = new JFrame ("Text Field font changer");
frame.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);
frame.setSize (400, 200);
frame.setLayout (new FlowLayout());

JTextField textField = new JTextField ("sample Text", 20);
textField.setFont (new Font ("Arial", Font.PLAIN, 14));
frame.add (textField);

JButton changeFontButton = new JButton ("change Font");
frame.add (changeFontButton);

changeFontButton.addActionListener (new ActionListener() {
@override
public void actionPerformed (ActionEvent e) {

textField.setFont (new Font ("Arial", Font.BOLD, 20));
textField.setForeground (Color.RED);
}
});
frame.setVisible (true);
}
}
```

O/P:




Q.NO.16 Explain how events are handled in java.

⇒

Event handling is a mechanism to control the events and to decide what should happen after an event occur. To handle the events, java follows the Delegation Event Model.

Delegation: Event Model

→ It has Source and Listeners.

* Source: Events are generated from the source. There are various sources like buttons, checkboxes, list, menu-item, choice, scrollbar, text components, windows etc. to generate events.

* Listeners: Listeners are used for handling the events generated from the source. Each of these listeners represents interfaces that are responsible for handling events.

To perform Event Handling, we need to register the source with the listener.

Different classes provide different registration methods.

Syntax:

addTypeListener() where Type represents the type of event.

Event Classes in Java

Some of the event class and their listener interface are given below:

| | Event Class | Listener Interface | Description |
|---|---|---|---|
| 1. | ActionEvent | ActionListener | An event that indicates that a component-defined action occurred like a button click or selecting an item from the menu-list. |
| 2. | KeyEvent | KeyListener | An event that occurs due to a sequence of keypresses on the keyboard. |
| 3. | MouseEvent | MouseListener & MouseMotionListener | The events that occur due to the user interaction with the mouse (Pointing Device). |
| 4. | WindowEvent | WindowListener | An event which indicates whether a window has changed its status or not. |

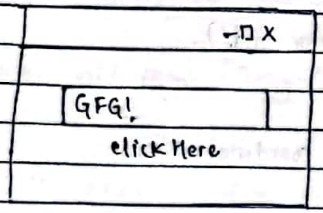
Different interfaces consists of different methods which are given below:

| | Listener interface | Methods |
|---|---|---|
| 1. | ActionListener | → actionPerformed() |
| 2. | MouseMotionListener | → mouseMoved()<br>→ mouseDragged() |
| 3. | MouseListener | → mousePressed()<br>→ mouseClicked()<br>→ mouseEntered()<br>→ mouseExited()<br>→ mouseReleased() |
| 4. | WindowListener | → windowActivated()<br>→ windowDeactivated()<br>→ windowOpened()<br>→ windowClosed()<br>→ windowClosing()<br>→ windowIconified()<br>→ windowDeiconified() |

Here's the basic program to demonstrate event handling in Java.

```
import java.awt.*;
import java.awt.event.*;

class GFGTop extends Frame implements ActionListener{

TextField textField;

GFGTop()
{
  textField = new TextField();
  textField.setBounds(60,50,180,25);
  Button button = new Button("click Here");
  button.setBounds(100,120,80,30);

  button.addActionListener(this);
  add(textField);
  add(Button);

  setVisible(true);
}

public void actionPerformed(ActionEvent e)
{
 textField.setText("GFG!");
}
public static void main(String[] args)
{
  new GFGTop();
}
}
```

O/P:




Q.NO.17 Define adapter class.

⇒

Every listener interface has an equivalent class which is called adapter class that provides default implementation of methods of corresponding interface.

The advantage of using adapter class is that we can provide override those methods which are necessary.

Q.NO.18 Write a program to display in a label whether the mouse pointer is in or outside a frame. Also display its x and y coordinates when the mouse is moved inside the frame.

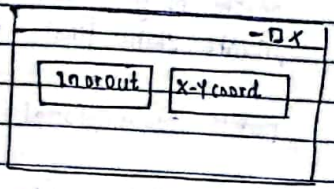
⇒

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

class Mouse implements MouseListener, MouseMotionListener{

JFrame f;
JLabel l1, l2;
public Mouse(){

f=new JFrame();
f.setSize(400,500);
```

```
l1 = new JLabel();
l1.setToolTipText ("In or out");

l2 = new JLabel();
l2.setToolTipText ("x-y coordinates");

f.add (l1);
f.add (l2);

f.setLayout (new FlowLayout ());
f.setVisible (true);

f.addMouseListener (this);
f.addMouseMotionListener (this);

}

public void mouseEntered (MouseEvent e)
{
  l1.setText ("In");
}

public void mouseExited (MouseEvent e)
{ l1.setText ("Out");
}

public void mouseMoved (MouseEvent e)
{
String loc = "X" + e.getX() + "Y" + e.getY();
 l2.setText(loc);
}
public void mouseClicked (MouseEvent e) {}
public void mousePressed (MouseEvent e) {}
public void mouseReleased (MouseEvent e) {}
```

Date
Page

```
public void mouseDragged (MouseEvent e) {}

public static void main (String [] args)
{
    new Mouse();        O.P:
}
}
```



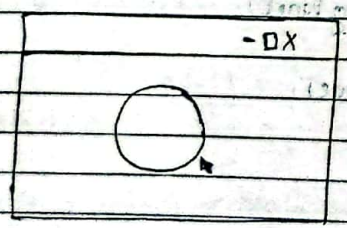

Q NO.19 Create an application to display a blue circle whenever the user clicks inside the frame.

⇒

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

public class BlueCircleOnClick {

public static void main (String [] args)
{
JFrame frame = new JFrame ("Blue Circle Drawer");
frame.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);
frame.setSize (600,400);

DrawingPanel drawingPanel = new DrawingPanel();
frame.add (drawingPanel);

frame.setVisible (true);
}
}
```

```
class DrawingPanel extends JPanel {

private int x = -50;
private int y = -50;
private static final int CIRCLE_DIAMETER = 50;

public DrawingPanel() {

addMouseListener(new MouseAdapter() {
@Override
public void mouseClicked(MouseEvent e) {

x = e.getX() - CIRCLE_DIAMETER/2;
y = e.getY() - CIRCLE_DIAMETER/2;
repaint();
    }
});
}
@Override
protected void paintComponent(Graphics g){
super.paintComponent(g);
g.setColor(Color.BLUE);
g.fillOval(x,y, CIRCLE_DIAMETER, CIRCLE_DIAMETER);
}
}
```

O/P:




Q.NO.20 Create a Popup Menu with some items. Display the title of the item which has been selected in a label.

=>

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

public class PopupMenuExample {

public static void main(String[] args) {

JFrame frame = new JFrame("Popup Menu Example");
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.setSize(400,300);
frame.setLayout(new BorderLayout());

JLabel label = new JLabel("Selected Item: None");
label.setHorizontalAlignment(SwingConstants.CENTER);
frame.add(label, BorderLayout.CENTER);

JPopupMenu popupMenu = new JPopupMenu();

String[] items = {"Item 1", "Item 2", "Item 3", "Item 4"};
for (String item : items) {

JMenuItem menuItem = new JMenuItem(item);
popupMenu.add(menuItem);

menuItem.addActionListener(new ActionListener() {
@Override
```

```
public void actionPerformed (ActionEvent e) {
    label.setText ("Selected Item:" + item);
    }
});
}
frame.addMouseListener (new MouseAdapter() {
@Override
    public void mouseReleased (MouseEvent e) {
    If (e.isPopupTrigger()) {
        popupMenu.show (e.getComponent(), e.getX(), e.getY());
    }
}
});

frame.setVisible (true);
}
}
```

Q No. 21 What are the features of JavaFx? Also Explain its architecture.

⇒ The features of JavaFx are described below:

1. FXML
   It is a declarative markup language that is used to create and design UI element.

2. SceneBuilder
   It allows us to create UI components & style them and generate FXML which can be portrailed to any IDE.

3. It provides high performance hardware accelerated graphics pipeline to render 2D and 3D elements.

4. It supports web view which enables us to embed html, CSS, JavaScript, SVG etc.

5. It allows CSS for styling UI elements.

6. It supports audio and video multimedia rendering with low latency.

7. MVC (Model View Control) pattern is supported.

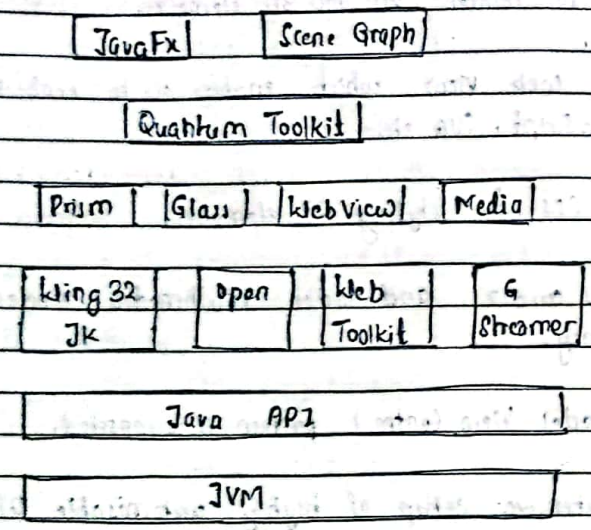
8. It has extensive setup of highly customisable UI elements.

9. Since it is a java library, it inherits all the properties of java by default.

10. Swing Interoperability
    Swing application can be converted into Java Application & JavaFx elements can be embedded into Swing application

11. Builtin UI control
    It has own set of UI control and is independent of underline operating system.

## JavaFx Architecture:




### Scene Graph:
→ It is the starting point of any JavaFx application
→ It is the hierarchial tree structure of nodes that represents all the visual element of UI.
→ It is also responsible for event handling.

### Quatum Toolkit
→ It combines prism and glass windowing toolkit and make them available to the upperlevel of stack.

### Prisms
→ It is a high performance hardware accelerated graphics pipeline that render 2D and 3D element Fx and animation.

### Glass
→ It acts as an interface between the javafx application and the native operating system.

### Webview:
→ It makes JavaFx application able to embedd web contents.
→ It use webtool kit (Open source browser)

### Media
→ It use open source GStreamer Engine to render audio and video multimedia.

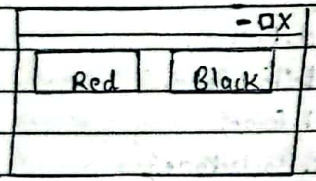
QNO.22 Write a program in JavaFx with two buttons named Black and Red. Change the background color of the scene to the respective color when a button is pressed.

⇒

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.StackPane;
import javafx.scene.paint.Color;
import javafx.scene.layout.BorderPane;
import javafx.stage.Stage;

public class BackgroundColorChanger extends Application {

@Override
public void start(Stage primaryStage) {

Button blackButton = new Button("Black");
Button redButton = new Button("Red");

BorderPane layout = new BorderPane();
layout.setTop(blackButton);
layout.setBottom(redButton);

Scene scene = new Scene(layout, 400,300);
```

```
blackButton.setOnAction(e -> layout.setStyle("-fx-background-color:
    black;"));
redButton.setOnAction(e -> layout.setStyle("-fx-background-color:
    red;"));

primaryStage.setTitle("Background Color Changer");
primaryStage.setScene(scene);
primaryStage.show();
}

public static void main(String[] args){
    launch(args);
}
}
```

O/P:




Q.NO.23 Explain the various layout managers available in JavaFx

⇒ Various types of layout manager available in JavaFx are described below:

① BorderPane

→ This layout allows the nodes to be placed in five different regions (top, bottom, left, right and center).

→ We can place a node in given directions using following methods:

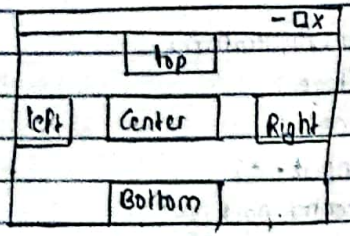
1. setTop();
2. setBottom();
3. setLeft();
4. setRight();
5. setCenter();

Sample program:

```
import javaFx.application.Application;
import javaFx.stage.Stage;
import javaFx.scene.Scene;
import javaFx.scene.input.*;
import javaFx.scene.control.package;
import javaFx.event.*; import javaFx.scene.layout.BorderPane;
class BorderDemo extends Application{

public void Start(Stage s) throws Exception{

Button top = new Button("Top");
Button bottom = new Button("Bottom");
Button left = new Button("Left");
Button Right = new Button("Right");
Button center = new Button("Center");

BorderPane root = new BorderPane();

root.setTop(top);
root.setBottom(bottom);
root.setLeft(left);
root.setRight(right);
root.setCenter(Center);

Scene scene = new Scene(root,400,300);
s.setScene(scene);
s.show();
}
```

psv main (String [] args)
{ launch (args);
}
}

O/P:




② HBox

→ This layout places the node horizontally in x-axis.
→ By default, there is no spacing betn the nodes.
→ We can provide spacing betn the node using following method:
↳ public void setSpacing (double space);

Sample program:

```
import javaFx.application.Application;
import javaFx.stage.Stage;
import javaFx.scene.Scene;
import javaFx.scene.input.*;
import javaFx.scene.control.package;
import javaFx.event.*;
import javaFx.scene.layout.HBox;
class HBoxDemo extends Application {
public void start (Stage s) throws Exception {
TextField t = new TextField();
Button b = new Button ("Click");
HBox root = new HBox();
root.setSpacing (5);
root.getChildren().addAll(t,b);
Scene scene = new Scene (root,300,300);
s.setScene (scene);
s.show();
}
psv main (String [] args)
{ launch (args);
}
```

③ VBox

→ This layout is used to place the nodes vertically.
→ By default, there is no spacing betn the nodes.
→ We can provide spacing betn the nodes using the method:
↳ public void setSpacing (double space);

Sample program:

```
import javaFx.application.Application;
import javaFx.stage.Stage;
import javaFx.scene.Scene;
import javaFx.scene.input.*;
import javaFx.scene.control.package;
import javaFx.event.*;
import javaFx.scene.layout.VBox;
class VBoxDemo extends Application {
public void start (Stage s) throws Exception {
TextField t = new TextField();
Button b = new Button ("click");
VBox root = new VBox();
root.setSpacing (5);
```

```
root.getChildren().addAll(t,b);
Scene scene = new Scene(root,300,300);
s.setScene(scene);
s.show();
}
p s v main(String[] args)
{
launch(args);
}
```

⑤ StackPane

→ This layout plows the nodes one on top of another as a stack.

Sample program:

```
import javaFx.application.Application;
import javaFx.stage.Stage;
import javaFx.scene.Scene;
import javaFx.scene.input.*;
import javaFx.scene.control.package;
import javaFx.event.*;
import javaFx.scene.layout.StackPane;
class StackDemo extends Application{

public void start(Stage s) throws Exception{
TextField t = new TextField();
Button b = new Button("click");
StackPane root = new StackPane();
root.getChildren().addAll(t,b);
Scene scene = new Scene(root,300,300);
s.setScene(scene);
s.show();
}
```

```
p s v main(String[] args)
{ launch(args);
}
```

⑤ GridPane

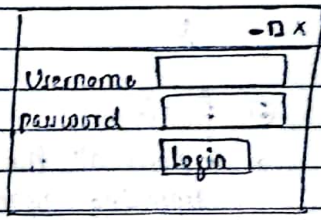
→ This layout allows us to place multiple nodes at multiple rows and columns.

→ By default, no spacing is provided between the nodes.

Methods:

```
public void vgap(double space);
public void hgap(double space);
public void gridLineVisible(boolean b)
```

Sample Program:




```
import javaFx.application.Application;
import javaFx.stage.Stage;
import javaFx.scene.Scene;
import javaFx.scene.input.*;
import javaFx.scene.control.package;
import javaFx.event.*;
import javaFx.scene.layout.GridPane;
class GridDemo extends Application{

public void start(Stage s) throws Exception{
Label u = new Label("User name");
Label p = new Label("Password");
TextField uname = new TextField();
PasswordField pwd = new PasswordField();
```

```
Button login = new Button ("Login");
GridPane root = new GridPane();

root.vgap(5);
root.hgap(10);
}

root.addColumn(0,u,p);
root.addColumn(1, uname, pwd, login);

Scene scene = new Scene(root, 400,400);
s.setScene(scene);
s.show();
}

p s v main (String [] args)
{
launch(args);
}
```

⑥ FlowPane

→ It places all the nodes either horizontally or vertically depending upon the orientation.

→ We can provide horizontal and vertical gap methods

* setVgap(Double space)
* setHgap(Double space)

Sample program:

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.control.Button;
import javafx.scene.layout.FlowPane;
import javafx.scene.Scene;

class Stack extends Application {
@override
public void start(Stage s) throws Exceptions {

Button b1 = new Button("1");
Button b2 = new Button("2");
FlowPane root = new FlowPane();
root.getChildren().addAll(b1,b2);
root.setHgap(10);
Scene scene = new Scene(root, 400,300);
s.setScene(scene);
s.show();
}
```