St. Peter's Engineering College (Autonomous)	Dept.	:	
Dullapally (P), Medchal, Hyderabad – 500100. **QUESTION BANK**	**Academic Year 2024-25**		

Subject Code	:	AS22 05PC02	Subject	:	Oops through Java			
Class/Section	:	B. Tech.	Year	:	II	Semester	:	III

BLOOMS LEVEL					
Remember	L1	Understand	L2	Apply	L3
Analyze	L4	Evaluate	L5	Create	L6

*****

Q. No	Question (s)	Marks	BL	CO
	UNIT - I			
1	a) **Define String.** A string is a series of characters stored in contiguous memory locations. It is terminated by a special character called the **null character**	**1M**	L2	C214.1
	b) **List out the Java buzzwords.** **1.**Object-Oriented, 2.Distributed, 3.Compiled and Interpreted, 4.Robust, 5.Secure, 6.Architecture-Neutral	**1M**	L1	C214.1

	c) **Define Class with a syntax.**
	public class MyClass {
	  // Data members (variables)
	  int myVariable; // Example variable
	
	  // Constructor (optional)
	  public MyClass() {
	    // Initialize variables or perform other setup
	  }
	
	  // Methods (functions)
	  public void myMethod() {
	    // Define behavior for this method
	  }
	}
		1M	L1	C214.1

	d) **Who is inventor of Java?** The **Java programming language** was developed by **James Gosling**,	**1M**	L1	C214.1
	e) **Define class and object.** A class serves as a blueprint or template for creating objects	**1M**	L1	C214.1

	An object is an instance of a class.			

2	**a) Discuss about polymorphism and its types with an example?**
	Compile-Time Polymorphism (Static Polymorphism):
	Achieved through function overloading.
	Function Overloading: When multiple functions have the same name but different parameters (number or type).
	Example:
	Java
	
	class Helper {
	   static int Multiply(int a, int b) {
	     return a * b;
	   }
	   static double Multiply(double a, double b) {
	     return a * b;
	   }
	}
	
	System.out.println(Helper.Multiply(2, 4)); // Output: 8
	System.out.println(Helper.Multiply(5.5, 6.3)); // Output: 34.65
	
	Run-Time Polymorphism (Dynamic Polymorphism):
	Resolved by the Java Virtual Machine (JVM) during runtime.
	Achieved through method overriding (when a subclass provides a specific implementation for a method defined in its superclass).
	Example:
	Java
	
	class Bicycle {
	   void ride() {
	     System.out.println("Riding a bicycle");
	   }
	}
	class MountainBike extends Bicycle {
	   void ride() {
	     System.out.println("Riding a mountain bike");
	   }
	}
	
	Bicycle bike1 = new MountainBike();
	bike1.ride(); // Output: "Riding a mountain bike"
		3M	L2	C214.1

	b) Discuss about Constructor and its types with an example? Constructor Basics: A constructor in Java is a special method that gets called when an object of a class is created. Constructors have the same name as the class and do not have a return type.	**3M**	L2	C214.1

They initialize the object's state (data members) and perform any necessary setup.
Constructors are essential for creating and initializing objects.
Types of Constructors:
Default Constructor:
A constructor with no parameters.
Automatically provided by Java if no other constructors are defined.
Example:
Java

```java
class Person {
    String name;

    // Default constructor (no parameters)
    Person() {
        name = "Unknown";
    }
}
```

Example:
Java

```java
class Student {
    String name;
    int age;

    // Parameterized constructor
    Student(String studentName, int studentAge) {
        name = studentName;
        age = studentAge;
    }
}
```

Example:
Java

```java
class Book {
    String title;

    // Copy constructor
    Book(Book originalBook) {
        title = originalBook.title;
    }
}

public class Main {
    public static void main(String[] args) {
        // Default constructor
        Person person = new Person();
```

	System.out.println("Person name: " + person.name);
	
	    // Parameterized constructor
	    Student student = new Student("Alice", 20);
	    System.out.println("Student name: " + student.name + ", Age: " + student.age);
	
	    // Copy constructor
	    Book originalBook = new Book();
	    originalBook.title = "Java Programming";
	    Book copiedBook = new Book(originalBook);
	    System.out.println("Copied book title: " + copiedBook.title);
	  }
	}			

	c) Discuss the use of class and object in Java with syntax and example?
	Class:
	A class serves as a blueprint or template for creating objects.
	It defines the state (fields or variables) and behavior (methods) that objects of that class will have.
	Think of a class as a sketch of a house – it describes the structure, but you need to build actual houses (objects) based on that sketch.
	Syntax for creating a class:
	Java
	
	class ClassName {
	   // Fields (variables)
	   // Methods
	}
	Example:
	Java
	
	class Bicycle {
	   private int gear = 5; // State (field)
	
	   public void braking() {
	      System.out.println("Working of Braking"); // Behavior (method)
	   }
	}
	
	Object:
	An object is an instance of a class.
	It represents a specific entity based on the class definition.
	For example, if Bicycle is a class, MountainBicycle, SportsBicycle, and TouringBicycle can be objects of that class.
	Creating an object:
	Java
		3M	L1	C214.1

Bicycle sportsBicycle = new Bicycle();
Bicycle touringBicycle = new Bicycle();
AI-generated code. Review and use carefully. More info on FAQ.
Here, sportsBicycle and touringBicycle are object names.
We use the new keyword along with the class constructor (which
has the same name as the class) to create objects.
Accessing Members:
You can use the object name followed by the dot (.) operator to
access fields and methods of the class.
Example:
Java

sportsBicycle.braking(); // Accessing the braking method
int gearValue = sportsBicycle.gear; // Accessing the gear field

d) Define Data encapsulation in Java with an example?
 	
Data encapsulation in Java is a fundamental concept in object-oriented programming (OOP). It involves bundling data (instance variables) and methods (functions) that operate on that data within a single unit (class). The goal is to hide the implementation details of a class from outside access and expose a public interface for interaction.
 	
Here's an example demonstrating Java encapsulation:
Java
 	
``java
class Person {
    private String name;
    private int age;
 	
    public String getName() {
        return name;
    }
 	
    public void setName(String name) {
        this.name = name;
    }
 	
    public int getAge() {
        return age;
    }
 	
    public void setAge(int age) {
        this.age = age;
    }
}
``
	3M	L1	C214.1

```
public class Main {
    public static void main(String[] args) {
        Person person = new Person();
        person.setName("John");
        person.setAge(30);

        System.out.println("Name: " +
person.getName());
        System.out.println("Age: " + person.getAge());
    }
}
```

				C214.1

e) **List out the various operator supports in Java with an example.**

f) **Arithmetic Operators**:
   a. These operators perform common mathematical operations.
   b. Examples:
      i. Addition: `int sum = 100 + 50;` (result: `sum` is 150)
      ii. Subtraction: `int difference = 100 - 50;` (result: `difference` is 50)
      iii. Multiplication: `int product = 10 * 5;` (result: `product` is 50)
      iv. Division: `double quotient = 20.0 / 3;` (result: `quotient` is approximately 6.6667)
      v. Modulus (remainder): `int remainder = 20 % 3;` (result: `remainder` is 2)

g) **Assignment Operators**:   **3M**   **L1**
   a. These operators assign values to variables.
   b. Example:
      i. `int x = 10;` (assigns the value 10 to variable `x`)
   c. Other assignment operators include `+=`, `-=` etc.

h) **Comparison Operators**:
   a. Used to compare values or variables.
   b. Examples:
      i. Equal to: `boolean isEqual = x == y;` (result: `isEqual` is `true` if `x` equals `y`)
      ii. Greater than: `boolean isGreater = x > y;` (result: `isGreater` is `true` if `x` is greater than `y`)

i) **Logical Operators**:
   a. Used to determine logic between variables or values.
   b. Examples:

	i. Logical AND (`&&`): `boolean result = (x < 5) && (y < 10);` (result: `result` is `true` if both conditions are true) ii. Logical OR (`||`): `boolean result = (x < 5) || (y < 10);` (result: `result` is `true` if at least one condition is true)			

3	**a) Explain in detail about operators with an example.**
	b) Arithmetic Operators:
	    a. These operators perform simple arithmetic operations on primitive data types:
	      i. `+`: Addition
	      ii. `-`: Subtraction
	      iii. `*`: Multiplication
	      iv. `/`: Division
	      v. `%`: Modulo (remainder)
	    b. Example:
	
	Java
	
	``java
	int a = 10;
	int b = 3;
	System.out.println("a + b = " + (a + b)); // Output: 13
	System.out.println("a - b = " + (a - b)); // Output: 7
	System.out.println("a * b = " + (a * b)); // Output: 30
	System.out.println("a / b = " + (a / b)); // Output: 3
	System.out.println("a % b = " + (a % b)); // Output: 1
	``
	
	AI-generated code. Review and use carefully. More info on FAQ.
	
	c) Unary Operators:
	    a. These operators work with a single operand:
	      i. `-`: Unary minus (negates the value)
	      ii. `+`: Unary plus (indicates positive value)
	      iii. `++`: Increment operator (post-increment and pre-increment)
	      iv. `--`: Decrement operator (post-decrement and pre-decrement)
	      v. `!`: Logical NOT (inverts a boolean value)
	    b. Example:
	
	Java
	
	``java
	int x = 10;
	``
		5M	L2	C214.1

```
int y = 10;
System.out.println("Postincrement: " + (x++)); //
Output: 10
System.out.println("Preincrement: " + (++x)); //
Output: 12
System.out.println("Postdecrement: " + (y--)); //
Output: 10
System.out.println("Predecrement: " + (--y)); //
Output: 8
```

AI-generated code. Review and use carefully. More info on FAQ.

d) **Assignment Operator**:
   a. The = operator assigns a value to a variable.
      Example:

Java

```
int result = 0;
result = a + b; // Assign the sum of a and b to
'result'
```

	b) What is meant by type casting and its types with a syntax?	**5M**	**L1**	C214.1
	Type casting in Java refers to converting a value from one data type to another. There are two main types of type casting:			

Widening Casting (Automatic):
   This occurs when you convert a smaller data type to a larger one.
   The order of widening casting is: `byte → short → char → int → long → float → double`.
   Example:

Java

```
int myInt = 9;
double myDouble = myInt; // Automatic casting: int to
double
```

AI-generated code. Review and use carefully. More info on FAQ.

Narrowing Casting (Manual):
   This involves converting a larger data type to a smaller one.
   You need to explicitly specify the target type using parentheses.
   The syntax is: `targetDataType variableName = (targetDataType) originalVariable;`

Example:

Java

```
double myDouble = 9.78;
int myInt = (int) myDouble; // Manual casting: double to int
```

Here's a real-life example where we calculate a user's percentage score in a game using type casting:

Java
```
int maxScore = 500;
int userScore = 423;
float percentage = (float) userScore / maxScore * 100.0f;
System.out.println("User's percentage is " + percentage);
```

c) What is meant by method overloading with an example.	**5M**	L1	

Method overloading in Java allows you to define multiple methods with the **same name**, but they **differ in parameters** (such as the number or types of parameters). Here's how it works:

Overloading by Changing the Number of Parameters:
You can create overloaded methods with different parameter counts.
Example:

Java

```
class Calculator {
    public static int add(int a, int b) {
        return a + b;
    }
    public static double add(double a, double b) {
        return a + b;
    }
}
```

Overloading by Changing the Data Type of Parameters:

Overloaded methods can accept different data types.
Example:

Java

```
class HelperService {
```

C214.1

```
    private String formatNumber(int value) {
        return String.format("%d", value);
    }
    private String formatNumber(double value) {
        return String.format("%.3f", value);
    }
    // Other overloaded methods...
```

e) Explain the characteristics of Java buzzwords.
f) Simple:
    a. Java is designed to be easy for both beginners and professional programmers.
    b. It removes unnecessary complexities, such as explicit pointers and operator overloading.
    c. If you're familiar with basic Object-Oriented Programming (OOP) concepts, Java's simplicity makes it accessible.
g) Object-Oriented:
    a. Java is a true object-oriented programming language.
    b. All code and data reside within objects and classes.
    c. The basic OOP concepts—like inheritance, polymorphism, abstraction, and encapsulation—are integral to Java.
h) Distributed:
    a. Java enables the creation of distributed applications across networks.
    b. It allows seamless access to remote objects on the Internet.
    c. Multiple programmers in different locations can collaborate on a single project.
i) Compiled and Interpreted:
    a. Java combines both compilation and interpretation.
    b. It compiles programs into an intermediate representation called Java Bytecode.
    c. The Bytecode is then interpreted by the Java Virtual Machine (JVM) to generate machine code.
j) Robust:
    a. Java emphasizes reliability and error handling.
    b. Strict typing checks code at compile time and runtime.
    c. Memory management is handled through garbage collection.
	5M	L2	C214.1

<table>
<tr><td></td><td>d. Exception handling captures serious errors.
k) **Secure**:
  a. Java provides a "firewall" between networked applications and your computer.
  b. It confines Java programs to the execution environment, preventing access to other parts of the system.</td><td></td><td></td><td></td></tr>
<tr><td></td><td>**a) Explain the principles of Object Oriented Programming.**

Object-Oriented Programming (OOP) is a powerful paradigm used in Java and other programming languages. Let's delve into the key principles of OOP in Java:

Abstraction:

o Definition: Abstraction simplifies complex systems by focusing on essential features while ignoring unnecessary details.

o In Java: You create abstract classes or interfaces that define common attributes and methods. These serve as blueprints for creating objects.

Encapsulation:

o Definition: Encapsulation bundles data (attributes) and methods (functions) into a single unit (class). It protects data by restricting direct access.

o In Java: You use access modifiers (public, private, protected) to control visibility. Private instance variables can only be accessed through public methods (getters and setters).

Inheritance:

o Definition: Inheritance allows creating new classes based on existing ones (parent-child relationship).

o In Java: Child classes (subclasses) inherit properties and behaviors from their parent class (superclass). It promotes code reuse and hierarchy.

Polymorphism:

Definition: Polymorphism enables treating objects of different</td><td>**5M**</td><td>L2</td><td>C214.1</td></tr>
</table>

	classes uniformly. o In Java: You achieve polymorphism through method overloading (same method name, different parameters) and method overriding (redefining a method in a subclass).			

4	a) **Explain the concepts of constructor and constructor overloading with an example.**
	the concepts of constructors and constructor overloading in Java, along with an example:
	
	Constructors:
	A constructor is a special method in a Java class that gets called when an object of that class is created.
	It initializes the object by setting its initial state (assigning values to instance variables).
	Constructors have the same name as the class and no return type (not even void).
	If you don't explicitly define a constructor, Java provides a default no-argument constructor.
	
	Constructor Overloading:
	Constructor overloading allows a class to have multiple constructors with different parameter lists.
	Each constructor can perform a different task based on the provided arguments.
	The compiler differentiates these constructors by considering the number and types of parameters.
	Example: Let's create a Box class with constructor overloading:
	Java
	
	``java
	class Box {
	   double width, height, depth;
	
	   // Constructor with three arguments
	   Box(double w, double h, double d) {
	      width = w;
	      height = h;
	      depth = d;
	   }
	
	   // Default constructor (no arguments)
	   Box() {
	      width = height = depth = 0;
	   }
	
	   // Constructor for a cube (one argument)
	   Box(double len) {
	      width = height = depth = len;
	``
	
		10M	L2	C214.1

```
  }

  double volume() {
     return width * height * depth;
  }
}

public class Test {
  public static void main(String args[]) {
     Box mybox1 = new Box(10, 20, 15);
     Box mybox2 = new Box(); // Default constructor
     Box mycube = new Box(7); // Cube constructor

     double vol;
     vol = mybox1.volume();
     System.out.println("Volume of mybox1 is " + vol);

     vol = mybox2.volume();
     System.out.println("Volume of mybox2 is " + vol);

     vol = mycube.volume();
     System.out.println("Volume of mycube is " + vol);
  }
}
```
output:
Volume of mybox1 is 3000.0
Volume of mybox2 is 0.0
Volume of mycube is 343.0

b) Define string. How to handle strings operations with an example.	**10M**	L2	

In Java, a string is a sequence of characters. It represents text data. For example, "hello" is a string containing the characters 'h', 'e', 'l', 'l', and 'o'. Strings are widely used in Java programming for tasks like storing user input, manipulating text, and more.

Creating a String:
There are two ways to create a string object in Java:

String Literal:
You can create a string using double quotes.
Example:
Java

String s = "welcome";
When you create a string literal, the JVM checks the string constant pool first. If the string already exists in the pool, a reference to the existing instance is returned. Otherwise, a new

C214.1

string instance is created and placed in the pool.
Using new Keyword:
You can create a string using the new keyword.
Example:
Java

String s = new String("Welcome");
AI-generated code. Review and use carefully. More info on FAQ.
In this case, a new string object is created in the normal heap
memory, and the literal "Welcome" is placed in the string constant
pool.
Common String Operations:
Concatenation:
You can concatenate strings using the + operator or the concat()
method.
Example:
Java

String firstName = "John";
String lastName = "Doe";
String fullName = firstName + " " + lastName;
Getting a Character by Index:
Use the charAt(index) method to retrieve a character at a specific
position.
Example:
Java

char firstChar = fullName.charAt(0); // Gets the first character ('J')
Replacing Substrings:
Use the replace(oldChar, newChar) or replace(oldStr, newStr)
methods.
Example:
Java

String modifiedName = fullName.replace("John", "Jane");
Finding Length:
Use the length() method to get the length of a string.
Example:
Java

int nameLength = fullName.length(); // Gets the length of the full
name

Splitting a String:
Use the split(delimiter) method to split a string into an array of
substrings.
Example:
Java

String[] parts = fullName.split(" "); // Splits by space			

c) What is meant by array? How can create an array in java with an example.
 	
An array in Java is a data structure that allows you to store multiple values of the same type in a single variable. It's like a collection of similar-typed variables accessed by a common name. Here are some key points about Java arrays:
 	
Fixed Size:
The number of values in a Java array is always fixed when you create it.
Once you define the size, it cannot change dynamically.
Declaration:
To declare an array, use the following syntax:
Java
 	
dataType[] arrayName;
AI-generated code. Review and use carefully. More info on FAQ.
dataType can be primitive types (e.g., int, char, double, etc.) or Java objects.
arrayName is an identifier.
Initialization:
You can initialize arrays during declaration:
Java
 	
int[] age = {12, 4, 5, 2, 5};
AI-generated code. Review and use carefully. More info on FAQ.
The Java compiler automatically determines the size based on the number of elements.
Accessing Elements:
Use the index number to access elements of an array:
Java
 	
System.out.println("First Element: " + age[0]);
System.out.println("Second Element: " + age[1]);
// ...
AI-generated code. Review and use carefully. More info on FAQ.
Example: Creating and Accessing an Array
Java
 	
public class ArrayExample {
   public static void main(String[] args) {
     // Declare and initialize an array
     int[] numbers = {1, 2, 3, 4, 5};
 	
     // Access array elements
     System.out.println("First Element: " + numbers[0]);
	10M	L1	

			C214.1

```
    System.out.println("Second Element: " + numbers[1]);
    // ...

    // Modify an element
    numbers[2] = 10;
    System.out.println("Modified Third Element: " +
numbers[2]);
    }
}
```
AI-generated code. Review and use carefully. More info on FAQ.
Output:

First Element: 1
Second Element: 2
Modified Third Element: 10

Q. No	Question (s)	Marks	BL	CO
	UNIT – II			
1	a) What is the use of final keyword in Java?	1M	L1	C214.2
	b) Differentiate Method overloading and overriding?	1M	L3	C214.2
	c) Define Package?	1M	L1	C214.2
	d) List out any two benefits of Inheritance.	1M	L1	C214.2
	e) List out the forms of Inheritance.	1M	L1	C214.1
2	a) Explain about Access Modifier?	3M	L1	C214.2
	b) Explain about single and multiple-level inheritance with an example?	3M	L2	C214.2
	c) List out the benefits and costs of Inheritance in Java?	3M	L2	C214.2
	d) Explain about generalization in Java?	3M	L2	C214.2
	e) Write a Java program to define final with inheritance concept?	3M	L1	C214.2
3	a) Define Interface. How to extending the interface with an example?	5M	L1	C214.2
	b) What is the difference between Interface and Class?	5M	L1	C214.2
	c) How to access package from another package?	5M	L2	C214.2
	d) What is the use of Super keyword? Give an example?	5M	L1	C214.2
	e) What is Abstraction in Java?	5M	L1	C214.2
4	a) Explain in detail about abstract class with an example.	10M	L2	C214.2
	b) What are the different forms of inheritance with an	10M	L1	C214.2

	example?			
	c) **Explain in detail about polymorphism and its types.**	**10M**	L2	C214.2

1. **What is the use of final keyword in Java?**   1m

a) The `final` keyword is a non-access modifier used for classes, attributes and methods, which makes them non-changeable (impossible to inherit or override).

b) The `final` keyword is useful when you want a variable to always store the same value, like PI (3.14159...).

c) The `final` keyword is called a "modifier".

2. **Differentiate Method overloading and overriding?**   1m

Method Overloading	Method Overriding
Method overloading is a compile-time polymorphism.	Method overriding is a run-time polymorphism.
Method overloading helps to increase the readability of the program.	Method overriding is used to grant the specific implementation of the method which is already provided by its parent class or superclass.
It occurs within the class.	It is performed in two classes with inheritance relationships.
Method overloading may or may not require inheritance.	Method overriding always needs inheritance.
In method overloading, methods must have the same name and different signatures.	In method overriding, methods must have the same name and same signature.

3. **Define Package?**   1m

Package in Java is a mechanism to encapsulate a group of classes, sub packages and interfaces. Packages are used for:
    Preventing naming conflicts. For example there can be two classes with name Employee in two packages, college.staff.cse.Employee and college.staff.ee.Employee

Making searching/locating and usage of classes, interfaces, enumerations and annotations easierProviding controlled access: protected and default have package level access control. A protected member is accessible by classes in the same package and its subclasses. A default member (without any access specifier) is accessible by classes in the same package only.

4. **List out any two benefits of Inheritance.** 1m

Inheritance in object-oriented programming (OOP) provides several benefits that help in code organization and reuse. Here are two key benefits:

1. **Code Reusability**:
   o **Description**: Inheritance allows you to create new classes that reuse the code of existing classes. This means you can define common functionality in a base class and extend it in derived classes without having to duplicate code.
   o **Example**: If you have a base class `Animal` with a method `makeSound()`, you can create derived classes like `Dog` and `Cat` that inherit from `Animal` and use the `makeSound()` method. This avoids the need to rewrite the `makeSound()` method for each derived class.
2. **Enhanced Maintainability**:
   o **Description**: Inheritance promotes a hierarchical organization of code, which makes it easier to maintain and update. Changes made to the base class automatically propagate to all derived classes, provided the changes don't break the contract established by the base class.
   o **Example**: If you update the `Animal` class to add a new method or modify an existing method, all classes derived from `Animal` will inherit these changes. This centralized update reduces the risk of inconsistencies and ensures that all derived classes benefit from improvements or bug fixes in the base class

5. **List out the forms of Inheritance** 1m

Single Inheritance: A class inherits from only one base class. This is the simplest form of inheritance.

Multiple Inheritances: A class inherits from more than one base class. This form of inheritance is not directly supported in Java due to ambiguity issues, but can be achieved through interfaces.

Multilevel Inheritance: A class inherits from a derived class, creating a chain of inheritance. This means a class acts as a base class for another class, which in turn acts as a base class for yet another class.

Hierarchical Inheritance: Multiple classes inherit from a single base class. This allows multiple subclasses to share common functionality from the base class.

Hybrid Inheritance :A combination of two or more types of inheritance. This can be complex and is not directly supported in some languages like Java due to potential ambiguity and complexity.

6. **Explain about Access Modifier?**   **3m**

   access modifiers control the visibility and accessibility of classes, methods, and variables. They define how and where the members of a class (such as methods and fields) can be accessed from other classes.
   `Public`**:** modifier allows the class, method, or field to be accessible from any other class in any package.
   `Protected:` modifier allows access to the class members from within the same package and from subclasses (even if they are in different packages). However, it does not allow access from non-subclass classes in other packages.
   `Private:` modifier restricts access to the class members so that they are only accessible within the same class. This is the most restrictive access level.
   Default : If no access modifier is specified, the default access level is applied. This level is also known as package-private. Members with default access are accessible only within the same package.

7. **Explain about single and multiple-level inheritance with an example?**   **3m**

   Single Inheritance: A class inherits from only one base class. This is the simplest form of inheritance.

   ```
   class Animal {
      void eat() {
         System.out.println("This animal eats food.");
      }
   }
   class Dog extends Animal {
      void bark() {
         System.out.println("The dog barks.");
      }
   }
   ```

   Multiple Inheritances: A class inherits from more than one base class. This form of inheritance is not directly supported in Java due to ambiguity issues, but can be achieved through interfaces

   ```
   interface Animal {
      void eat();
   }
   interface Pet {
      void play();
   }

   class Dog implements Animal, Pet {
      public void eat() {
         System.out.println("The dog eats.");
      }
   ```

```
    public void play() {
        System.out.println("The dog plays.");
    }
}
```

8. List out the benefits and costs of Inheritance in Java?   3m

Benefits of Inheritance:

1. **Code Reusability**
2. **Ease of Maintenance**:
3. **Hierarchical Classification**:
4. **Polymorphism**:
5. **Encapsulation**:

Costs of Inheritance:
1. Tight Coupling
2. Increased Complexity
3. Fragile Base Class Problem
4. Inheritance Overuse
5. Performance Overheads

9. Explain about generalization in Java?   3m

generalization refers to the concept of designing a more general or abstract class from which more specific subclasses can be derived. This process is a key aspect of object-oriented design and plays a crucial role in abstraction, code reusability, and hierarchy management. Here's an in-depth look at generalization:

Key Aspects of Generalization:
1. Abstract Classes and Methods
2. **Super class and Subclass Relationships**:
3. **Polymorphism**:
4. **Code Reusability**:
5. Design Flexibility

10. Write a Java program to define final with inheritance concept?   3M

the `final` keyword can be used with classes, methods, and variables, and it plays a significant role in inheritance. Here's how it applies to each:

- **`final` class**: A class declared as `final` cannot be subclassed. This means no other class can extend a `final` class.
- **`final` method**: A method declared as `final` cannot be overridden by subclasses. This means that the behavior of the method is fixed and cannot be changed.
- **`final` variable**: A variable declared as `final` can only be assigned once. This makes the variable a constant

```
// Define a final class
```

```java
final class Vehicle {

    // Final variable

    private final String brand;

    // Constructor to initialize the final variable

    Vehicle(String brand) {

        this.brand = brand;

    }

    // Final method

    final void displayInfo() {

        System.out.println("Vehicle brand: " + brand);

    }

    // Getter for brand

    public String getBrand() {

        return brand;

    }

}

// Attempt to extend the final class (This will cause a compile-time error)

// class Car extends Vehicle {

//     // This code will not compile because Vehicle is final

// }


public class Main {

    public static void main(String[] args) {

        // Create an instance of the final class Vehicle
```

Vehicle myVehicle = new Vehicle("Toyota");

// Display vehicle information

myVehicle.displayInfo();

// Accessing final variable through getter method

System.out.println("Brand accessed through getter: " + myVehicle.getBrand());

}

}

11. Define Interface. How to extending the interface with an example?   5M

interface is a reference type that can contain only constants, method signatures, default methods, static methods, and nested types. Interfaces are used to specify a set of methods that a class must implement, without providing the method implementations. Interfaces are a way to achieve abstraction and multiple inheritance in Java.

Key Points About Interfaces

1. **Method Signatures**: An interface defines a contract by specifying method signatures (method names, return types, and parameters) but does not provide implementations for these methods.
2. **Implementation**: A class implements an interface and provides concrete implementations for all of its methods.
3. **Multiple Inheritance**: Interfaces support multiple inheritance, meaning a class can implement multiple interfaces.
4. **Default Methods**: Java 8 introduced default methods, which allow interfaces to provide method implementations.
5. **Static Methods**: Interfaces can also have static methods with implementations.

Extending Interfaces

When you extend an interface, you create a new interface that inherits the methods of the parent interface. An interface can extend multiple other interfaces. This is similar to extending classes but involves multiple inheritance of method signatures.

Example of Extending Interfaces

Here's a detailed example to illustrate how to define and extend interfaces in Java:

```java
Copy code
```

```java
// Define the base interface
interface Animal {
    void eat(); // Abstract method
    void sleep(); // Abstract method
}

// Define another interface that extends the base interface
interface DomesticAnimal extends Animal {
    void play(); // Additional method
}

// Implement the extended interface in a class
class Dog implements DomesticAnimal {
    @Override
    public void eat() {
        System.out.println("The dog eats.");
    }

    @Override
    public void sleep() {
        System.out.println("The dog sleeps.");
    }

    @Override
    public void play() {
        System.out.println("The dog plays.");
    }
}

// Main class to test the implementation
public class Main {
    public static void main(String[] args) {
        // Create an instance of Dog
        DomesticAnimal myDog = new Dog();

        // Call methods implemented in the Dog class
        myDog.eat();
        myDog.sleep();
        myDog.play();
    }
}
```

Output: The dog eats. The dog sleeps. The dog plays

12. What is the difference between Interface and Class?   5M

Class	Interface
The keyword used to create a class is "class"	The keyword used to create an interface is "interface"

Class	Interface
A class can be instantiated i.e., objects of a class can be created.	An Interface cannot be instantiated i.e. objects cannot be created.
Classes do not support multiple inheritance.	The interface supports multiple inheritances.
It can be inherited from another class.	It cannot inherit a class.
It can be inherited by another class using the keyword 'extends'.	It can be inherited by a class by using the keyword 'implements' and it can be inherited by an interface using the keyword 'extends'.
It can contain constructors.	It cannot contain constructors.
It cannot contain abstract methods.	It contains abstract methods only.

13. How to access package from another package?   5m

Accessing a package from another package in Java involves importing the classes or interfaces from the desired package into the class where you want to use them. Here are the general steps to achieve this in Java:

1. **Structure Your Project**: Ensure your project is structured correctly. For example:

project/
├── src/
├── package1/
│   ├── MyClass1.java
├── package2/
├── MyClass2.java
2. **Declare the Packages**: In your Java files, declare the packages at the top of each file.
// Inside src/package1/MyClass1.java
package package1;

public class MyClass1 {
public void display() {
System.out.println("Hello from MyClass1 in package1");
}

```
}
```

// Inside src/package2/MyClass2.java
package package2;

import package1.MyClass1;

public class MyClass2 {
   public static void main(String[] args) {
     MyClass1 obj = new MyClass1();
     obj.display();
   }
}
3.Compile the Packages: Compile the packages using the Java compiler (`javac`).
Navigate to the `src` directory and compile the classes:
javac package1/MyClass1.java
javac package2/MyClass2.java
4.Run the Program: Run the program by specifying the fully qualified name of the class
with the `java` command:
java package2.MyClass2

14. What is the use of Super keyword? Give an example?   5m

The `super` keyword in Java is used to refer to the immediate parent class object. It can be used
for various purposes, such as calling parent class methods, accessing parent class constructors,
and accessing parent class fields when they are hidden by subclasses.

Here are the primary uses of the `super` keyword:

1. **Accessing Parent Class Constructor**: The `super()` call can be used to invoke the
   constructor of the parent class.
2. **Accessing Parent Class Methods**: The `super` keyword can be used to call methods of
   the parent class that are overridden in the subclass.
3. **Accessing Parent Class Fields**: The `super` keyword can be used to access fields of the
   parent class when they are hidden by fields in the subclass.

// Parent class
class Animal {
   String name;

   // Constructor of parent class
   Animal(String name) {
     this.name = name;
   }

   // Method in parent class
   void display() {

```
        System.out.println("I am an animal. My name is " + name);
    }
}

// Child class
class Dog extends Animal {
    String breed;

    // Constructor of child class
    Dog(String name, String breed) {
        // Call the constructor of the parent class
        super(name);
        this.breed = breed;
    }

    // Method in child class
    void display() {
        // Call the display method of the parent class
        super.display();
        System.out.println("I am a dog. My breed is " + breed);
    }
}

// Main class
public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog("Buddy", "Golden Retriever");
        dog.display();
    }
}
```

15. What is Abstraction in Java?   5m

Abstraction in Java is a fundamental concept of object-oriented programming that involves hiding the complex implementation details of a system and exposing only the necessary and relevant parts to the user. It helps in reducing complexity and allows the programmer to focus on interacting with the objects at a higher level.

Key Points about Abstraction in Java:

1. **Abstract Classes**: These are classes that cannot be instantiated on their own and can include abstract methods (methods without a body) that must be implemented by subclasses.
2. **Interfaces**: These are a collection of abstract methods that any class can implement, providing a way to achieve full abstraction (as interfaces cannot have any method implementations in Java 7 and earlier; Java 8 introduced default and static methods).

   abstract class Animal {

```
   // Abstract method (does not have a body)
   abstract void sound();

   // Regular method
   void sleep() {
      System.out.println("This animal is sleeping.");
   }
}

class Dog extends Animal {
   // Provide implementation of the abstract method
   void sound() {
      System.out.println("Woof");
   }
}

public class Main {
   public static void main(String[] args) {
      Dog dog = new Dog();
      dog.sound(); // Output: Woof
      dog.sleep(); // Output: This animal is sleeping.
   }
}
```

16. Explain in detail about abstract class with an example   10m

An abstract class in Java is a class that cannot be instantiated directly and is intended to be subclassed. It can contain abstract methods (methods without an implementation) as well as concrete methods (methods with an implementation). The abstract methods must be implemented by subclasses, ensuring that the subclasses provide specific behaviors while the abstract class can provide common functionality.

Key Points about Abstract Classes

1. **Cannot be Instantiated**: You cannot create an instance of an abstract class directly. It must be subclassed.
2. **Can Have Abstract Methods**: These methods do not have a body and must be implemented by the subclasses.
3. **Can Have Concrete Methods**: These methods have an implementation and can be inherited by subclasses.
4. **Can Have Constructors**: Even though you cannot instantiate an abstract class, you can have constructors in it, which are called when a subclass is instantiated.

Example

Here is a detailed example to illustrate the concept of abstract classes:

```
// Abstract class
abstract class Animal {
```

```java
    String name;

    // Constructor
    Animal(String name) {
        this.name = name;
    }

    // Abstract method (no implementation)
    abstract void makeSound();

    // Concrete method
    void sleep() {
        System.out.println(name + " is sleeping.");
    }
}

// Subclass of Animal
class Dog extends Animal {
    // Constructor
    Dog(String name) {
        super(name);
    }

    // Implementation of the abstract method
    @Override
    void makeSound() {
        System.out.println(name + " says: Woof!");
    }
}

// Another subclass of Animal
class Cat extends Animal {
    // Constructor
    Cat(String name) {
        super(name);
    }

    // Implementation of the abstract method
    @Override
    void makeSound() {
        System.out.println(name + " says: Meow!");
    }
}

// Main class to test the abstract class and its subclasses
public class Main {
    public static void main(String[] args) {
        // Create instances of Dog and Cat
        Dog dog = new Dog("Buddy");
        Cat cat = new Cat("Whiskers");

        // Call the methods
        dog.makeSound(); // Output: Buddy says: Woof!
        dog.sleep();     // Output: Buddy is sleeping.

        cat.makeSound(); // Output: Whiskers says: Meow!
        cat.sleep();     // Output: Whiskers is sleeping.
    }
}
```

17. What are the different forms of inheritance with an example?
    10m

nheritance in Java is a key aspect of object-oriented programming, allowing classes to inherit properties and methods from other classes. Here are the different forms of inheritance in Java along with examples:

1. **Single Inheritance**: A class inherits from one base class.

    ```java
    class Animal {
      void sound() {
        System.out.println("Some sound");
      }
    }

    class Dog extends Animal {
      void sound() {
        System.out.println("Bark");
      }
    }

    public class Main {
      public static void main(String[] args) {
        Dog dog = new Dog();
        dog.sound();  // Output: Bark
      }
    }
    ```

2. **Multilevel Inheritance**: A class inherits from a derived class, forming a chain.

    ```java
    class Animal {
      void sound() {
        System.out.println("Some sound");
      }
    }

    class Dog extends Animal {
      void sound() {
        System.out.println("Bark");
      }
    }

    class Puppy extends Dog {
      void sound() {
        System.out.println("Yap");
      }
    }
    ```

```java
public class Main {
    public static void main(String[] args) {
        Puppy puppy = new Puppy();
        puppy.sound();  // Output: Yap
    }
}
```

3. **Hierarchical Inheritance**: Multiple classes inherit from a single base class.

```java
class Animal {
    void sound() {
        System.out.println("Some sound");
    }
}

class Dog extends Animal {
    void sound() {
        System.out.println("Bark");
    }
}

class Cat extends Animal {
    void sound() {
        System.out.println("Meow");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        Cat cat = new Cat();
        dog.sound();  // Output: Bark
        cat.sound();  // Output: Meow
    }
}
```

4. **Multiple Inheritance (using interfaces)**: Java does not support multiple inheritance with classes to avoid the diamond problem. However, multiple inheritance is possible using interfaces.

```java
interface Animal {
    void sound();
}

interface Vehicle {
    void move();
}

class RobotDog implements Animal, Vehicle {
    public void sound() {
        System.out.println("Beep");
    }
```

```
    public void move() {
       System.out.println("Rolling");
    }
}

public class Main {
    public static void main(String[] args) {
       RobotDog robotDog = new RobotDog();
       robotDog.sound();  // Output: Beep
       robotDog.move();   // Output: Rolling
    }
}
```

5. **Hybrid Inheritance**: Java does not directly support hybrid inheritance due to the lack of support for multiple inheritance with classes. However, a combination of interfaces and classes can achieve similar results.

```
interface Animal {
    void sound();
}

interface Fish {
    void swim();
}

class Bird implements Animal {
    public void sound() {
       System.out.println("Chirp");
    }
}

class FlyingFish extends Bird implements Fish {
    public void swim() {
       System.out.println("Swimming");
    }
}

public class Main {
    public static void main(String[] args) {
       FlyingFish flyingFish = new FlyingFish();
       flyingFish.sound();  // Output: Chirp
       flyingFish.swim();   // Output: Swimming
    }
}
```

18. **Explain in detail about polymorphism and its types.**   **10m**

Polymorphism is a core concept in object-oriented programming (OOP) that allows objects of different classes to be treated as objects of a common super class. It enables one interface to be used for a general class of actions. The specific action is determined by the exact nature of the situation.

There are two main types of polymorphism in OOP: **Compile-time polymorphism** and **Run-time polymorphism**.

1. Compile-time Polymorphism (Static Binding or Method Overloading)

Compile-time polymorphism is achieved through method overloading and operator overloading. This type of polymorphism is resolved during compile time.

Method Overloading: Method overloading occurs when multiple methods in the same class have the same name but different parameters (different type or number of parameters).

```
class MathOperation {
  // Method with 2 int parameters
  int add(int a, int b) {
     return a + b;
  }

  // Method with 3 int parameters
  int add(int a, int b, int c) {
     return a + b + c;
  }

  // Method with 2 double parameters
  double add(double a, double b) {
     return a + b;
  }
}

public class Main {
  public static void main(String[] args) {
     MathOperation operation = new MathOperation();
     System.out.println(operation.add(2, 3));        // Output: 5
     System.out.println(operation.add(2, 3, 4));     // Output: 9
     System.out.println(operation.add(2.5, 3.5));    // Output: 6.0
  }
}
```

2.Run-time Polymorphism (Dynamic Binding or Method Overriding)

Run-time polymorphism is achieved through method overriding. This type of polymorphism is resolved during run time.

Method Overriding: Method overriding occurs when a subclass provides a specific implementation for a method that is already defined in its superclass. The method in the subclass should have the same name, return type, and parameters as the method in the superclass.

```java
class Animal {

    void sound() {

        System.out.println("Animal makes a sound");

    }

}

class Dog extends Animal {

    @Override

    void sound() {

        System.out.println("Dog barks");

    }

}

class Cat extends Animal {

    @Override

    void sound() {

        System.out.println("Cat meows");

    }

}


public class Main {

    public static void main(String[] args) {

        Animal myAnimal = new Animal();

        Animal myDog = new Dog();
```

Animal myCat = new Cat();

myAnimal.sound(); // Output: Animal makes a sound

myDog.sound();   // Output: Dog barks

myCat.sound();   // Output: Cat meows

    }

}

Types of Polymorphism

1. **Compile-time Polymorphism (Static Binding or Method Overloading)**
   o Resolved during compile time.
   o Achieved through method overloading.
   o Multiple methods with the same name but different signatures (parameter list).
2. **Run-time Polymorphism (Dynamic Binding or Method Overriding)**
   o Resolved during runtime.
   o Achieved through method overriding.
   o Methods in subclass override methods in superclass with the same signature.

Q. No	Question (s)	Marks	BL	CO
	UNIT – III			
1	**a) What is Exception handling?**	**1M**	L1	C214.3
	b) What are all the keywords required for defining exception handling?	**1M**	L1	C214.3
	c) Name one method from the String class that is used to concatenate strings?	**1M**	L1	C225.3
	d) List out the some pre-defined exceptions?	**1M**	L1	C214.3
	e) Write the syntax of try block?	**1M**	L1	C214.1
2	**a) How the exceptions are handled in exception handling with an example?**	**3M**	L1	C214.3
	b) Write a Java program with nested try block in the exception handling?	**3M**	L1	C214.3
	c) Explain Arithmetic Exception, IO Exception, and NullPointer Exception, with syntax and example	**3M**	L2	C214.3
	d) How finally block used in the Java application?	**3M**	L1	C214.3

	e) What is meant by runtime exception?	3M	L1	C214.3
3	a) What is meant by exception handling?	5M	L1	C214.3
	b) How to implement the nested try block in the exception handling mechanism?	5M	L2	C214.3
	c)) Explain Daemon Thread with an example?	5M	L2	C214.3
	d) Differentiate between Multitasking and Multithreading?	5M	L2	C214.3
	e) Explain String Handling with proper example?	5M	L2	C214.3
4	a)What is package? Explain about built-in packages in java?	10M	L3	C214.3
	b) Explain the thread life cycle in detail?	10M	L2	C214.3
	c) Explain the Java thread by Extending thread class and implementing Runnable interfaces with an example?	10M	L2	C214.3

a) What is Exception handling?
Ans: Exception handling is a programming technique used to manage errors and exceptional conditions that occur during the execution of a program. It allows a program to continue running or terminate gracefully instead of crashing. This is achieved by using constructs like try, catch, throw, throws, and finally to detect and handle exceptions in a controlled manner.

b) **What are all the keywords required for defining exception handling?**
Ans:
The keywords required for defining exception handling in Java are:
try: Used to specify a block of code that might throw an exception.
catch: Used to handle the exception that occurs in the associated try block.
finally: Used to define a block of code that will always be executed after the try block, regardless of whether an exception was thrown or not.
throw: Used to explicitly throw an exception.
throws: Used in method signatures to declare that a method can throw exceptions.

c) **Name one method from the String class that is used to concatenate strings?**
Ans: One method from the String class used to concatenate strings is concat(). For example, str1.concat(str2) joins str2 to the end of str1.

d) **List out the some pre-defined exceptions?**
Ans: Any three are sufficient :
Here are some pre-defined exceptions in Java:

ArithmeticException: Thrown when an arithmetic operation, such as division by zero, occurs.
ArrayIndexOutOfBoundsException: Thrown when an array has been accessed with an illegal index.
NullPointerException: Thrown when an application attempts to use null in a case where an object is required.
ClassCastException: Thrown when an attempt is made to cast an object to a subclass of which it is not an instance.
IllegalArgumentException: Thrown to indicate that a method has been passed an illegal or inappropriate argument.
IllegalStateException: Thrown to signal that a method has been invoked at an illegal or inappropriate time.
IndexOutOfBoundsException: Thrown to indicate that an index of some sort is out of range.

NumberFormatException: Thrown to indicate that an attempt to convert a string to a numeric type failed.

IOException: Thrown when an I/O operation has failed or been interrupted.

FileNotFoundException: Thrown when an attempt to open the file denoted by a specified pathname has failed.

e) **Write the syntax of try block?**

Ans: try {
   // Code that might throw an exception
} catch (ExceptionType1 e1) {
   // Code to handle ExceptionType1
} catch (ExceptionType2 e2) {
   // Code to handle ExceptionType2
} finally {
   // Code that will always execute, regardless of whether an exception is thrown or not
}

Explanation:

- The try block contains code that might throw an exception.
- The catch blocks handle specific exceptions that are thrown in the try block.
- The finally block contains code that executes after the try block, regardless of whether an exception was thrown or caught. The finally block is optional.

2a) **How the exceptions are handled in exception handling with an example?**

Ans:

Exceptions in Java are handled using a combination of try, catch, and optionally finally blocks. Here's an example to illustrate how exceptions are handled:

```
public class ExceptionHandlingExample {
   public static void main(String[] args) {
      try {
         // Code that may throw an exception
         int[] numbers = {1, 2, 3};
         System.out.println("The fourth number is: " + numbers[3]);
      } catch (ArrayIndexOutOfBoundsException e) {
         // Code to handle the exception
         System.out.println("An error occurred: " + e.getMessage());
      } finally {
         // Code that will always execute
         System.out.println("This is the finally block.");
      }

      System.out.println("Rest of the program continues...");
   }
}
```

Here, when the exception is thrown, the control transfers to the catch block, the error message is printed, and then the finally block executes. The program then continues executing the code after the try-catch-finally construct.

2 b) Write a Java program with nested try block in the exception handling?

Ans: public class NestedTryExample {
   public static void main(String[] args) {
     try {
       // Outer try block
       System.out.println("Outer try block started.");

       try {
         // Inner try block
         System.out.println("Inner try block started.");
         int result = 10 / 0; // This will throw ArithmeticException
         System.out.println("This line will not be executed.");
       } catch (ArithmeticException e) {
         // Inner catch block
         System.out.println("Inner catch block: " + e.getMessage());
       } finally {
         // Inner finally block
         System.out.println("Inner finally block executed.");
       }

       // Code after inner try-catch-finally
       System.out.println("Code after inner try-catch-finally.");

     } catch (Exception e) {
       // Outer catch block
       System.out.println("Outer catch block: " + e.getMessage());
     } finally {
       // Outer finally block
       System.out.println("Outer finally block executed.");
     }

     System.out.println("Rest of the program continues...");
   }
}

Explanation:

Outer try block: Starts execution of the outer block of code.

Inner try block: Nested within the outer try block. It contains code that may throw an exception. In this case, int result = 10 / 0; will throw an ArithmeticException.

Inner catch block: Catches exceptions thrown by the inner try block. It handles the ArithmeticException and prints an appropriate message.

Inner finally block: Executes after the inner try and catch blocks, regardless of whether an exception was thrown or caught.

Outer catch block: Catches any exceptions that are not handled by the inner catch block. In this example, it won't catch anything since the inner catch block already handled the exception.

Outer finally block: Executes after the outer try block, regardless of whether an exception was thrown or not.

2 c) Explain Arithmetic Exception, IO Exception, and NullPointer Exception, with syntax and example?

Ans: explanation of ArithmeticException, IOException, and NullPointerException with syntax and examples:

1. ArithmeticException

Description: This exception is thrown when an exceptional arithmetic condition occurs, such as division by zero.

Example:

```
public class ArithmeticExceptionExample {
    public static void main(String[] args) {
        try {
            int result = 10 / 0; // This will throw ArithmeticException
        } catch (ArithmeticException e) {
            System.out.println("Caught an ArithmeticException: " + e.getMessage());
        }
    }
}
```

IOException:

```
import java.io.FileReader;
import java.io.IOException;

public class IOExceptionExample {
    public static void main(String[] args) {
        try {
            FileReader file = new FileReader("nonexistentfile.txt"); // This will throw IOException
        } catch (IOException e) {
            System.out.println("Caught an IOException: " + e.getMessage());
        }
    }
}
```

NullPointerException:

```
public class NullPointerExceptionExample {
    public static void main(String[] args) {
        String str = null;
        try {
            int length = str.length(); // This will throw NullPointerException
        } catch (NullPointerException e) {
            System.out.println("Caught a NullPointerException: " + e.getMessage());
        }
    }
}
```

2 d) How finally block used in the Java application?

Ans: In Java, the finally block is used to ensure that a certain piece of code is always executed, regardless of whether an exception was thrown or not. This is useful for cleanup activities such as closing files, releasing resources, or performing any other necessary final steps.

Synatx:

```
try {
    // Code that may throw an exception
} catch (ExceptionType e) {
    // Code to handle the exception
} finally {
    // Code that will always execute
```

}
Point to remember:
1. Always Executes: The finally block will execute whether an exception is thrown or not. Even if the try or catch block contains a return statement, the finally block will still be executed.
2. Resource Cleanup: It is commonly used for closing resources like files or database connections that need to be closed regardless of whether an exception occurred.
3. Optional: The finally block is optional. If it is not included, you simply omit it.
e) What is meant by runtime exception?
Ans: A runtime exception in Java refers to exceptions that are thrown during the execution of a program, as opposed to compile-time exceptions that are checked by the compiler. These are a type of unchecked exceptions, meaning they are not required to be caught or declared in the method signature.

Key Characteristics:
Unchecked Exceptions: Runtime exceptions are a subclass of RuntimeException, which is itself a subclass of Exception. They do not need to be declared in the throws clause of a method or be caught in a try-catch block.

Occurrence: These exceptions occur at runtime, often due to programming bugs or issues with the code logic, such as trying to access an array element with an invalid index or performing arithmetic operations with invalid values.

Common Examples:

NullPointerException: Thrown when an application attempts to use a null reference where an object is required.
ArrayIndexOutOfBoundsException: Thrown when an array has been accessed with an illegal index.
ArithmeticException: Thrown when an arithmetic operation, such as division by zero, occurs.
ClassCastException: Thrown when an attempt is made to cast an object to a subclass of which it is not an instance.
Example:

```
public class RuntimeExceptionExample {
    public static void main(String[] args) {
        try {
            // Example of NullPointerException
            String str = null;
            System.out.println(str.length()); // This will throw NullPointerException

            // Example of ArrayIndexOutOfBoundsException
            int[] numbers = {1, 2, 3};
            System.out.println(numbers[5]); // This will throw ArrayIndexOutOfBoundsException
        } catch (Exception e) {
            // Catching all exceptions (including runtime exceptions)
            System.out.println("Caught an exception: " + e.getMessage());
        }

        System.out.println("Program continues...");
```

```
    }
}
```
3 a) What is meant by exception handling with proper example with program?
Ans: Definition:

Exception handling in Java is a mechanism to manage runtime errors, allowing a program to continue executing or terminate gracefully instead of crashing. It involves using specific constructs to detect, catch, and handle exceptions that occur during the execution of a program. The primary constructs used are try, catch, finally, and throw.

Components of Exception Handling:

try Block: Contains code that may throw an exception. It is used to specify a block of code in which exceptions might occur.
catch Block: Handles the exception thrown by the try block. It catches the exception and provides a response or message.
finally Block: Contains code that will always execute, regardless of whether an exception was thrown or caught. It's typically used for cleanup activities.
throw Statement: Used to explicitly throw an exception.
throws Keyword: Declares that a method can throw one or more exceptions. It is used in the method signature.

```
Example Program: public class ExceptionHandlingDemo {
    public static void main(String[] args) {
        int[] numbers = {1, 2, 3};

        try {
            // Code that may throw exceptions
            System.out.println("Accessing element at index 2: " + numbers[2]); // Valid access

            System.out.println("Accessing element at index 5: " + numbers[5]); // This will throw
ArrayIndexOutOfBoundsException

            int result = 10 / 0; // This will throw ArithmeticException

        } catch (ArrayIndexOutOfBoundsException e) {
            // Handling ArrayIndexOutOfBoundsException
            System.out.println("Caught ArrayIndexOutOfBoundsException: " + e.getMessage());
        } catch (ArithmeticException e) {
            // Handling ArithmeticException
            System.out.println("Caught ArithmeticException: " + e.getMessage());
        } finally {
            // Code that will always execute
            System.out.println("This is the finally block. It executes regardless of whether an
exception was thrown or not.");
        }

        System.out.println("Program continues executing after exception handling.");
    }
```

}

Here is a Java program demonstrating exception handling by handling different types of exceptions including ArithmeticException, ArrayIndexOutOfBoundsException, and using the finally block for resource cleanup.

3 b) How to implement the nested try block in the exception handling mechanism?
Ans: In Java, nested try blocks allow you to handle exceptions at multiple levels, providing more granular control over error handling. You can nest try blocks within each other to catch and handle exceptions at different levels of the code.

Here's how to implement nested try blocks in the exception handling mechanism:

Structure of Nested try Blocks
Outer try Block: The outermost try block surrounds a section of code that might throw an exception.
Inner try Block: Nested within the outer try block, this block contains code that may also throw exceptions. The inner try block can have its own catch blocks and finally block.
Catch Blocks: Each try block can have its own catch blocks to handle exceptions thrown by the respective try block.
Finally Blocks: Each try block can have its own finally block, which will execute regardless of whether an exception was thrown or caught.
Give one example:---

3 c) Explain Daemon Thread with an example?
Ans: Daemon Threads in Java are special types of threads that run in the background and do not prevent the JVM from exiting. They are typically used for tasks that should run continuously in the background, such as garbage collection or monitoring services. The primary characteristic of daemon threads is that they do not block the termination of the JVM when all non-daemon threads have finished execution.

Key Characteristics of Daemon Threads:
Background Tasks: Daemon threads are often used for background tasks that need to run as long as the application is running but should not prevent the application from shutting down.
JVM Termination: The JVM will exit when all non-daemon threads have finished execution, regardless of whether daemon threads are still running.
Set Daemon Status: Threads can be set to daemon status using the setDaemon(true) method before they are started.

```
Example: public class DaemonThreadExample {
   public static void main(String[] args) {
      // Creating a non-daemon thread
      Thread mainThread = Thread.currentThread();
      System.out.println("Main thread: " + mainThread.getName());

      // Creating a daemon thread
      Thread daemonThread = new Thread(() -> {
         try {
            while (true) {
               System.out.println("Daemon thread running...");
               Thread.sleep(1000); // Sleep for 1 second
```

```
      }
   } catch (InterruptedException e) {
       System.out.println("Daemon thread interrupted.");
   }
});

// Set daemon status to true
daemonThread.setDaemon(true);

// Start the daemon thread
daemonThread.start();

// Main thread sleeps for 5 seconds and then exits
try {
   Thread.sleep(5000); // Sleep for 5 seconds
} catch (InterruptedException e) {
   System.out.println("Main thread interrupted.");
}

System.out.println("Main thread exiting...");
   }
}
```

3 d) Differentiate between Multitasking and Multithreading?

Ans:

	multitasking	**Multithreading**
1	In multitasking, users are allowed to perform many tasks by CPU	While in multithreading, many threads are created from a process through which computer power is increased.
2	Multitasking involves often CPU switching between the tasks.	While in multithreading also, CPU switching is often involved between the threads
3	In multitasking, the processes share separate memory	While in multithreading, processes are allocated the same memory.
4	The multitasking component involves multiprocessing	While the multithreading component does not involve multiprocessing.
5	In multitasking, the CPU is provided in order to execute many tasks at a time.	While in multithreading also, a CPU is provided in order to execute many threads from a process at a time.
6	In multitasking, processes don't share the same resources, each process is allocated separate resources.	While in multithreading, each process shares the same resources
7	Multitasking is slow compared to multithreading	While multithreading is faster.
8	In multitasking, termination of a process takes more time	While in multithreading, termination of thread takes less time.
9	Involves running multiple independent processes or tasks	Involves dividing a single process into multiple threads that can execute concurrently

10	Examples: running multiple applications on a computer, running multiple servers on a network	Examples: splitting a video encoding task into multiple threads, implementing a responsive user interface in an application

3 e) Explain String Handling with proper example?

STRING HANDLING
☐ A string is a sequence of characters surrounded by double quotations. In a java programming language, a string is the object of a built-in class **String**.

☐ The string created using the **String** class can be extended. It allows us to add more characters after its definition, and also it can be modified.

Example
String siteName = "javaprogramming"; siteName = "javaprogramminglanguage";
String handling methods
In java programming language, the String class contains various methods that can be used to handle string data values.
The following table depicts all built-in methods of String class in java.
S.No Method Description
1 charAt(int) Finds the character at given index

2 length() Finds the length of given string

3 compareTo(String) Compares two strings

4 compareToIgnoreCase(String) Compares two strings, ignoring case
5 concat(String) Concatenates the object string with argument string.

6 contains(String) Checks whether a string contains sub-string

7 contentEquals(String) Checks whether two strings are same

8 equals(String) Checks whether two strings are same

9 equalsIgnoreCase(String) Checks whether two strings are same, ignoring case

10 startsWith(String) Checks whether a string starts with the specified string

11 isEmpty() Checks whether a string is empty or not

12 replace(String, String) Replaces the first string with second string

Replaces the first string with second string at all
replaceAll(String, String)
13
substring(int, int)
14
occurrences.
Extracts a sub-string from specified start and end index values
15 toLowerCase() Converts a string to lower case letters

16 toUpperCase() Converts a string to upper case letters

17 trim() Removes whitespace from both ends

18 toString(int) Converts the value to a String object

Example program:

```
public class JavaStringExample
{
public static void main(String[] args)
{
String title = "Java Programming";
String siteName = "String Handling Methods";
 System.out.println("Length of title: " + title.length());
System.out.println("Char at index 3: " + title.charAt(3));
System.out.println("Index of 'T': " + title.indexOf('T'));
System.out.println("Empty: " + title.isEmpty());
 System.out.println("Equals: " + siteName.equals(title));
System.out.println("Sub-string: " + siteName.substring(9, 14));
System.out.println("Upper case: " + siteName.toUpperCase());
}
}
```

4 a) Explain the thread life cycle in detail?
Ans: Life Cycle of a Thread
Threads, the smallest unit of a process, have a life cycle. In Java, this life cycle features six main states that any thread can occupy at a given point in time:

1. New
A thread is in this state when you've created an instance of the Thread class but haven't invoked the start() method yet. It remains in this state until the program starts the thread.

2. Active
This state consists of two sub-states, Runnable and Running. Runnable implies that the thread is ready for execution and is waiting for resource allocation by the thread scheduler. Running means the thread scheduler has selected the thread and is currently executing its run() method.

3. Blocked / Waiting
A thread enters this state when it is temporarily inactive and waiting for a signal to proceed due to reasons like waiting for a resource to become available (Blocked) or waiting for another thread to perform a specific action (Waiting).
4. Timed Waiting
In this state, a thread is waiting for a specified period. A thread might enter this state through methods like Thread.sleep(long millis) or Object.wait(long timeout) where it waits for a particular duration before resuming its activities.
5. Terminated
This is the final state in the thread life cycle. The thread arrives here when it has completed its execution, i.e., its run() method has been completed, or it has been abruptly terminated due to an unhandled exception. Once in this state, the thread cannot be resumed.
As Java developers, having a profound understanding of these states and the transitioning nuances between them provides us with the power to harness threads effectively, optimizing the execution of

Fig: State Transition Diagram of a Thread
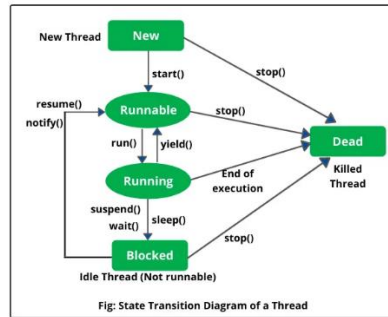
our concurrent programs.

b) **Explain the Java thread by Extending thread class and implementing Runnable interfaces with an example?**

Ans:

Aspect	Extending Thread Class	Implementing Runnable Interface
Inheritance	Extends Thread class, which is not possible if your class already extends another class	Implements Runnable interface, allowing for multiple inheritance of classes
Flexibility	Less flexible due to single inheritance limitation	More flexible as it allows for implementation of other interfaces or inheritance from other classes
Code Separation	Code for task execution and thread management is combined	Code for task execution is separated from thread management, promoting cleaner design
Reusability	Limited reusability if you need to extend another class	Can reuse the Runnable implementation with different Thread instances
Example Usage	Suitable for simple cases where a class needs to perform threading	Preferred for complex scenarios where threading and business logic are better separated

Example for both has to be given

```java
class MyThread extends Thread {
    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println("Thread " + Thread.currentThread().getId() + " is running, i=" + i);
            try {
                Thread.sleep(500); // Sleep for 500 milliseconds
            } catch (InterruptedException e) {
                System.out.println("Thread interrupted.");
            }
        }
    }
}

public class Main {
```

```
    public static void main(String[] args) {
       MyThread thread1 = new MyThread();
       MyThread thread2 = new MyThread();

       thread1.start(); // Start thread1
       thread2.start(); // Start thread2
    }
 }
Example :
class MyRunnable implements Runnable {
    @Override
    public void run() {
       for (int i = 0; i < 5; i++) {
          System.out.println("Runnable " + Thread.currentThread().getId() + " is running, i=" + i);
          try {
             Thread.sleep(500); // Sleep for 500 milliseconds
          } catch (InterruptedException e) {
             System.out.println("Runnable interrupted.");
          }
       }
    }
 }

 public class Main {
    public static void main(String[] args) {
       MyRunnable myRunnable = new MyRunnable();

       Thread thread1 = new Thread(myRunnable);
       Thread thread2 = new Thread(myRunnable);

       thread1.start(); // Start thread1
       thread2.start(); // Start thread2
    }
 }
```

Q. No	Question (s)	Marks	BL	CO
	UNIT – IV			
1	a) **Define 'Events' in Java with respect to event handling.** **Ans:** Changing the state of an object is known as an event.	**1M**	L2	C214.5

	b) **What are 'Event sources' in Java?** Ans: A source is an object that generates an event. Sources may generate more than one type of event. A source must register. listeners in order for the listeners to receive notifications about a specific type of event.	**1M**	L1	C214.5
	c) **What is an event listener?** **Ans:** It is also known as event handler. Listener is responsible for generating response to an event.	**1M**	L1	C214.5
	d) **What is a dialog box?** **Ans:** A Dialog is a top-level window with a title and a border that is typically used to take some form of input from the user.	**1M**	L1	C214.5
	e) **What is a layout manager?** **Ans:** A layout manager is an object that implements the LayoutManager interface* and determines the size and position of the components within a container	**1M**	L1	C214.5
2	a) **Explain different types of 'Event sources' in Java?**	**3M**	L2	C214.5
	b) **Explain 'Event Listeners' and their role in Java.**	**3M**	L2	C214.5
	c) **Describe the different ways to handle events in Java.**	**3M**	L2	C214.5
	d) **Explain about Handling Mouse Events.**	**3M**	L2	C214.5
	e) **Explain about "Button" component.**	**3M**	L2	C214.5
3	a) **Explain Delegation Event Model in Java.**	**5M**	L2	C214.5
	b) **Discuss the advantages and disadvantages of using adapter classes.**	**5M**	L2	C214.5
	c) **Explain Handling Mouse Events in java with example.**	**5M**	L2	C214.5
	d) **Write a GUI program in Java that containing a Button labeled "Click Here." On clicking the button, the program should display "Button clicked!" in the console.**	**5M**	L2	C214.5
	e) **What is the Difference Between TextField and TextArea in Java**	**5M**	L1	C214.5
4	a) **Describe the hierarchy of AWT user interface components.**	**10M**	L2	C214.5
	b) **Explain the complete event handling mechanism in Java, covering event sources, listeners, event classes, and their interactions in detail.**	**10M**	L2	C214.5
	c) **Discuss in detail 5 types of Layout Managers.**	**10M**	L2	C214.5

UNIT-4 Question Bank Answers:

2.a) Explain different types of 'Event sources' in Java?

S.No.	Event Class	Listener Interface	Methods
1.	ActionEvent	ActionListener	actionPerformed()
2.	AdjustmentEvent	AdjustmentListener	adjustmentValueChang
3.	ComponentEvent	ComponentListener	componentResized() componentMoved() componentShown() a componentHidden(
4.	ContainerEvent	ContainerListener	componentRemoved() componentAdded(
5.	FocusEvent	FocusListener	focusLost() and focusGained()
6.	ItemEvent	ItemListener	itemStateChanged(
7.	KeyEvent	KeyListener	keyPressed(), keyReleased(), and keyTyped().
8.	MouseEvent	MouseListener and MouseMotionListener	mouseClicked(), mousePressed(), mouseEntere mouseExited() and mouseReleased() are the mouseListener methods. mouseDregged() and

			mouseMoved() are the MouseMotionListener() metho...
9.	**MouseWheelEvent**	**MouseWheelListener**	**mouseWheelMoved(**
10.	**TextEvent**	**TextListener**	**textChanged()**
11.	**WindowEvent**	**WindowListener**	**windowActivated(),** **windowDeactivated(** **windowOpened(),** **windowClosed(),** **windowClosing(),** **windowIconfied() an** **windowDeiconified(**

2.b) Explain 'Event Listeners' and their role in Java.

Ans: Event Listener Interfaces in Java:
Listeners are created by implementing one or more of the interfaces defined by
the java.awt.event package. When an event occurs, the event source invokes the appropriate
method defined by the listener and provides an event object as its argument**.**

Component Listener Interface in Java:

This interface defines four methods that are invoked when a component is resized, moved,
shown, or hidden. Their general forms are shown here:
1. void component Resized(Component Event ce)
2. void component Moved(ComponentEvent ce)
3. void component Shown(ComponentEvent ce)
4. void component Hidden(ComponentEvent ce)

KeyListener Interface in Java:
This interface defines three methods. The **keyPressed( )** and **keyReleased( )** methods are
invoked when a key is pressed and released, respectively. The **keyTyped()** method is invoked
when a character has been entered. The general forms of these methods are shown here:
1. void keyPressed(KeyEvent ke)
2. void keyReleased(KeyEvent ke)
3. void keyTyped(KeyEvent ke

Mouse Listener Interface in Java:

This interface defines five methods. If the mouse is pressed and released at the same point, **mouseClicked( )** is invoked. When the mouse enters a component, the **mouseEntered( )** method is called. When it leaves, **mouseExited( )** is called. The **mousePressed( )** and **mouseReleased()** methods are invoked when the mouse is pressed and released, respectively. The general forms of these methods are shown here:

1. void mouseClicked(MouseEvent me)
2. void mouseEntered(MouseEvent me)
3. void mouseExited(MouseEvent me)
4. void mousePressed(MouseEvent me)
5. void mouseReleased(MouseEvent me**)**

2. c) Describe the different ways to handle events in Java.

Ans: Event Handling in Java
An event can be defined as changing the state of an object or behavior by performing actions. Actions can be a button click, cursor movement, keypress through keyboard or page scrolling, etc.
The java.awt.event package can be used to provide various event classes.
Classification of Events
- Foreground Events
- Background Events
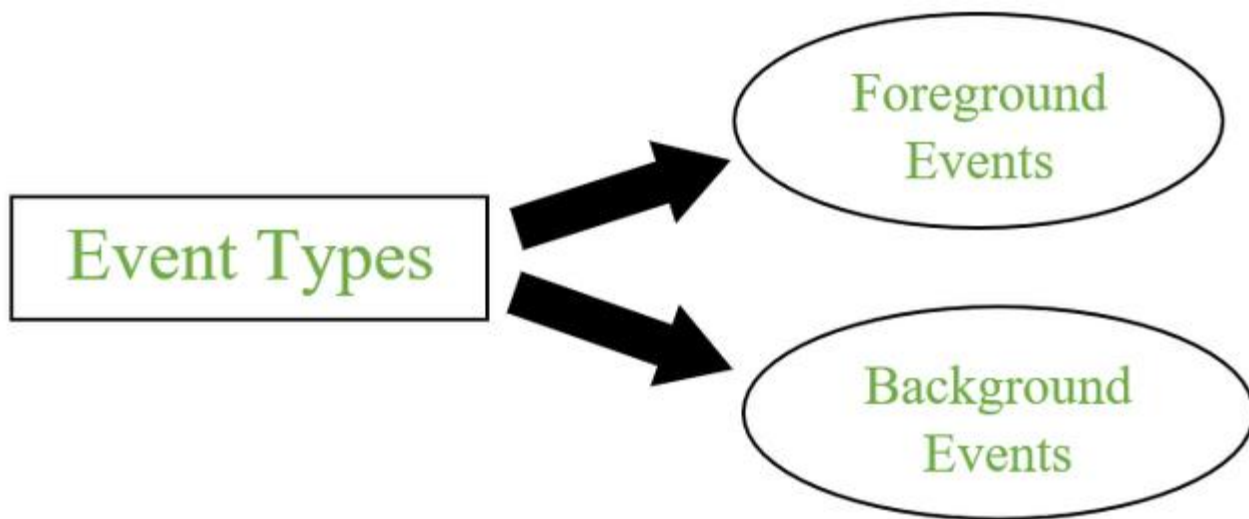
Types of Events:
1. Foreground Events
Foreground events are the events that require user interaction to generate, i.e., foreground events are generated due to interaction by the user on components in Graphic User Interface (GUI). Interactions are nothing but clicking on a button, scrolling the scroll bar, cursor moments, etc.

2. Background Events

Events that don't require interactions of users to generate are known as background events. Examples of these events are operating system failures/interrupts, operation completion, etc.

Event Handling

It 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.*

2.d) Explain about Handling Mouse Events.

Ans:

MOUSE EVENT HANDLING

Mouse events can be trapped for any GUI component that derives from java.awt.Component. The methods of interfaces **MouseListener** and **MouseMotionListener** are summarized below

Methods of Interface **MouseListener**

public void mousePressed(MouseEvent event)
Called when a mouse button is pressed with the mouse cursor on a component.

public void mouseClicked(MouseEvent event)
Called when a mouse button is pressed and released on a component without moving the mouse cursor

public void mouseReleased(MouseEvent event)
Called when a mouse button is released after being pressed. This event is always preceded by a mousePressed event.

public void mouseEntered(MouseEvent event)
    Called when the mouse cursor enters the bounds of a component.

public void mouseExited(MouseEvent event)
    Called when the mouse cursor leaves the bounds of a component.

Methods of Interface **MouseMotionListener**

public void mouseDragged(MouseEvent event)
Called when the mouse button is pressed with the mouse cursor on a component and the mouse is moved. This event is always preceded by a call to **mousePressed**

public void mouseMoved(MouseEvent event)
Called when the mouse is moved with the mouse cursor on a component.

2 .e) Explain about "Button" component.

A button is basically a control component with a label that
generates an event when pushed. The Button class is used to create
a labeled button that has platform independent implementation.
The application result in some action when the button is pushed.

S. no.	Constructor	Description
1.	Button( )	It constructs a new button with an empty string i.e. it has no label.
2.	Button (String text)	It constructs a new button with given string as its label.

Button Class Methods

Sr. no.	Method	Description
1.	void setText (String text)	It sets the string message on the button
2.	String getText()	It fetches the String message on the button.
3.	void setLabel (String label)	It sets the label of button with the specified string.
4.	String getLabel()	It fetches the label of the button.
5.	void addNotify()	It creates the peer of the button.
6.	AccessibleContext getAccessibleContext()	It fetched the accessible context associated with the button.
7.	void addActionListener(ActionListener l)	It adds the specified action listener to get the action events from the button.
8.	String getActionCommand()	It returns the command name of the action event fired

		by the button.
9.	ActionListener[ ] getActionListeners()	It returns an array of all the action listeners registered on the button.
10.	T[ ] getListeners(Class listenerType)	It returns an array of all the objects currently registered as FooListeners upon this Button.
11.	protected String paramString()	It returns the string which represents the state of button.
12.	protected void processActionEvent (ActionEvent e)	It process the action events on the button by dispatching them to a registered ActionListener object.
13.	protected void processEvent (AWTEvent e)	It process the events on the button
14.	void removeActionListener (ActionListener l)	It removes the specified action listener so that it no longer receives action events from the button.
15.	void setActionCommand(String command)	It sets the command name for the action event given by the button.

```java
import java.awt.*;

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

// create instance of frame with the label
Frame f = new Frame("Button Example");

// create instance of button with label
    Button b = new Button("Click Here");
    // set the position for the button in frame
      b.setBounds(50,100,80,30);

      // add button to the frame
      f.add(b);
    // set size, layout and visibility of frame
      f.setSize(400,400);
      f.setLayout(null);
```
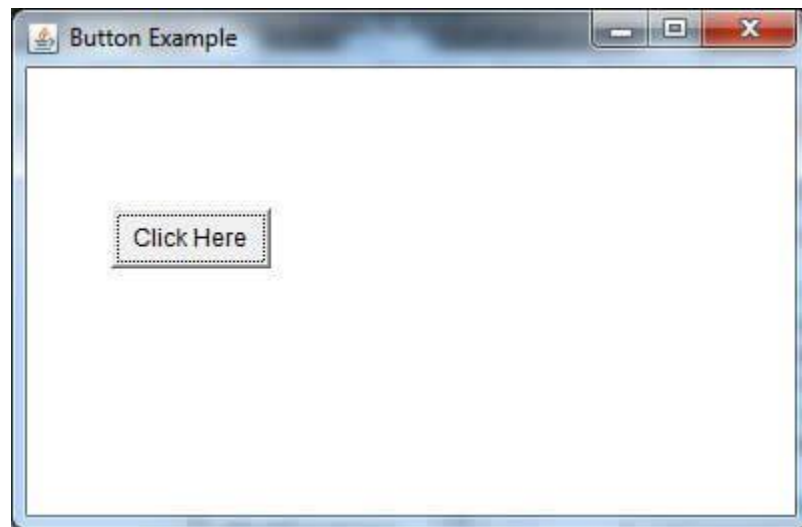
      f.setVisible(true);
    }
    }

      To compile the program using command prompt type the following commands
C:\Users\Anurati\Desktop\abcDemo>javac ButtonExample.java

Output:

3.a) Explain Delegation Event Model in Java.

Ans:

DELEGATION EVENT MODEL IN JAVA

- The Delegation Event model is defined to handle events in GUI programming languages.
- The GUI stands for Graphical User Interface, where a user graphically/visually interacts with the system.
- The GUI programming is inherently event-driven; whenever a user initiates an activity such as a mouse activity, clicks, scrolling, etc., each is known as an event that is mapped to a code to respond to functionality to the user. This is known as event handling.

The below image demonstrates the event processing.
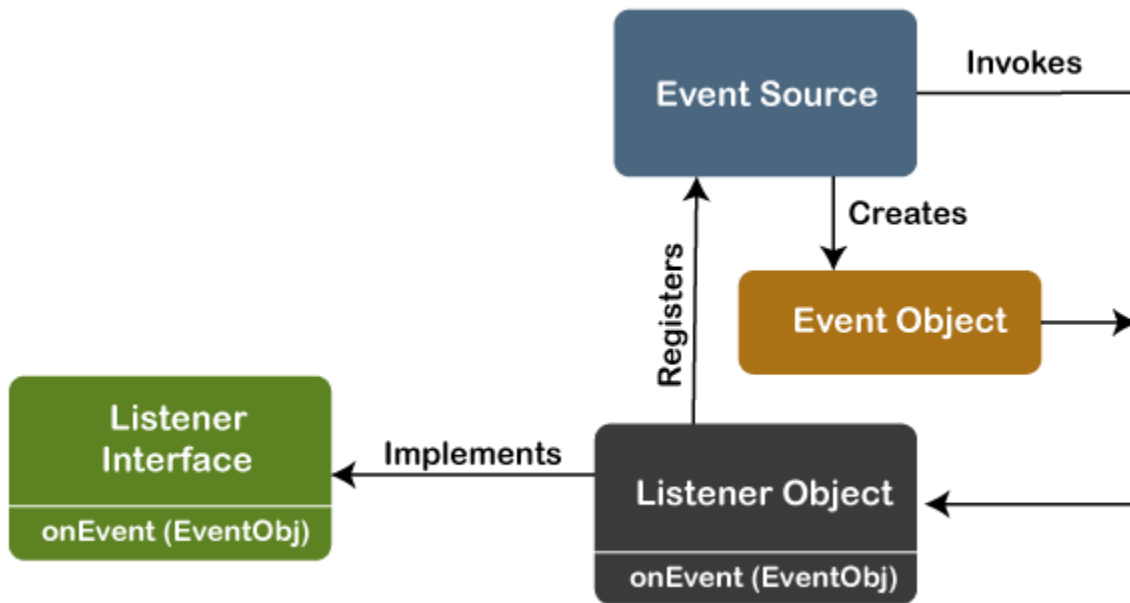
- In this model, a source generates an event and forwards it to one or more listeners.
- The listener waits until it receives an event. Once it receives the event, it is processed by the listener and returns it.

REGISTRATION METHODS

For registering the component with the Listener, many classes provide the registration methods.

Button

```
public void addActionListener(ActionListener a)
{
}
```

MenuItem

```
public void addActionListener(ActionListener a)
{
}
```

TextField

```
public void addActionListener(ActionListener a)
{
}
public void addTextListener(TextListener a)
{
```

```
    }
```

TextArea

```
    public void addTextListener(TextListener a)

    {

    }
```

Checkbox

```
    public void addItemListener(ItemListener a)
    {
    }
```

Choice

```
    public void addItemListener(ItemListener a)
    {
    }
```

List

```
    public void addActionListener(ActionListener a)
    {
    }
    public void addItemListener(ItemListener a)
    {
    }
```

3.b) Discuss the advantages and disadvantages of using adapter classes.
Ans: In JAVA, an adapter class allows the default implementation of listener interfaces. The notion of listener interfaces stems from the Delegation Event Model. It is one of the many techniques used to handle events in Graphical User Interface (GUI) programming languages, such as JAVA**.**

Advantages of an Adapter Class
1. An adapter class assists unrelated classes in working together.
2. With the use of an adapter class, the same class can be used in multiple ways.
3. Users are provided with the option of a pluggable kit for developing applications. Class usage, therefore, becomes highly reusable.
4. It enhances the transparency of classes.
5. It provides a way to include related patterns in a class.

   Disadvantage:

1.Adapter is a Structural Design Pattern that allows incompatible interfaces between classes to work together without modifying their source code.

3.c) Explain Handling Mouse Events in java with example?
Ans:

Java MouseListener Interface:

The Java Mouse Listener is notified whenever you change the state of mouse. It is notified against Mouse Event. The MouseListener interface is found in java.awt.event package. It has five methods.

Methods of MouseListener interface:
The signature of 5 methods found in MouseListener interface are given below:
1. **public abstract void** mouseClicked(MouseEvent e);
2. **public abstract void** mouseEntered(MouseEvent e);
3. **public abstract void** mouseExited(MouseEvent e);
4. **public abstract void** mousePressed(MouseEvent e);
5. **public abstract void** mouseReleased(MouseEvent e);

Java MouseListener Example
```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.  public class MouseListenerExample extends Frame implements MouseListener{
4.      Label l;
5.      MouseListenerExample(){
6.          addMouseListener(this);
7.
8.          l=new Label();
9.          l.setBounds(20,50,100,20);
10.         add(l);
11.         setSize(300,300);
12.         setLayout(null);
13.         setVisible(true);
14.     }
15.     public void mouseClicked(MouseEvent e) {
16.         l.setText("Mouse Clicked");
17.     }
18.     public void mouseEntered(MouseEvent e) {
19.         l.setText("Mouse Entered");
20.     }
21.     public void mouseExited(MouseEvent e) {
22.         l.setText("Mouse Exited");
23.     }
24.     public void mousePressed(MouseEvent e) {
25.         l.setText("Mouse Pressed");
26.     }
27.     public void mouseReleased(MouseEvent e) {
28.         l.setText("Mouse Released");
29.     }
30. public static void main(String[] args) {
31.     new MouseListenerExample();
32. }
33. }
```
Output:

Mouse Entered
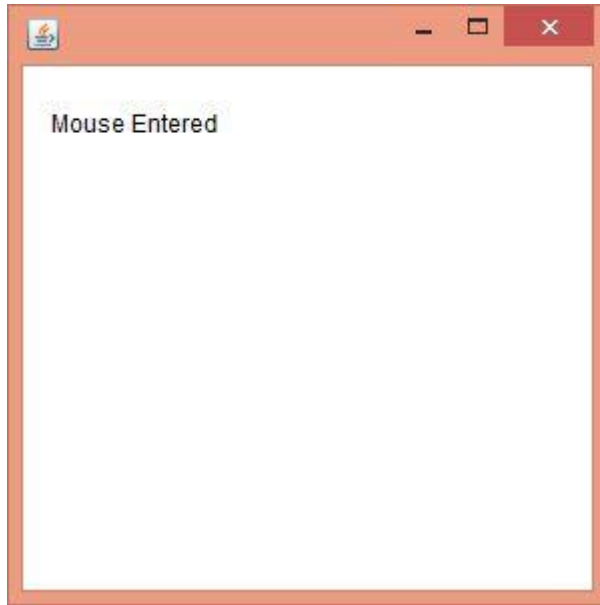
3.d) Write a GUI program in Java that containing a Button labeled "Click Here." On clicking the button, the program should display "Button clicked!" in the console.

The class **JButton** is an implementation of a push button. This component has a label and generates an event when pressed. It can also have an Image.

Class Declaration
Following is the declaration for **javax.swing.JButton** class −
public class JButton
  extends AbstractButton
    implements Accessible

Java AWT Button

A button is basically a control component with a label that generates an event when pushed. The Button class is used to create a labeled button that has platform independent implementation. The application result in some action when the button is pushed.

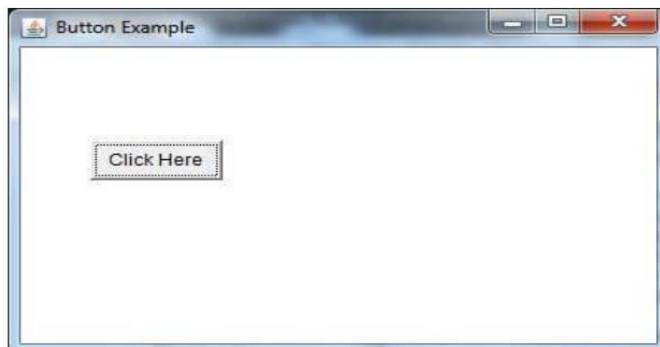## Example

```
import java.awt.*;
public class ButtonExample
{`
  public static void main (String[] args)
  {
          Frame f = new Frame("Button Example");  Button b = new
          Button("Click Here");  b.setBounds(50,100,80,30);
```

```
        f.add(b);  f.setSize(400,400);
        f.setLayout(null);  f.setVisible(true);
    }
}
```

Output:

3. e) What is the Difference Between Text Field and Text Area in Java.

The main difference between JTextField and JTextArea in Java is that a JTextField allows entering a single line of text in a GUI application while the JTextArea allows entering multiple lines of text in a GUI application.
JTextField

- A JTextFeld is one of the most important components that allow the user to an input text value in a single line format.
- A JTextField will generate an ActionListener interface when we trying to enter some input inside it.
- The JTextComponent is a superclass of JTextField that provides a common set of methods used by JTextfield.
- The important methods in the JTextField class are setText(), getText(), setEnabled(), etc.

Example:
```
import javax.swing.*;
import java.awt.*;
public class JTextFieldTest {
  public static void main(String[] args) {
    final JFrame frame = new JFrame("JTextField Demo");
    JLabel lblFirstName = new JLabel("First Name:");
    JTextField tfFirstName = new JTextField(20);
    lblFirstName.setLabelFor(tfFirstName);
    JLabel lblLastName = new JLabel("Last Name:");
    JTextField tfLastName = new JTextField(20);
    lblLastName.setLabelFor(tfLastName);
    JPanel panel = new JPanel();
    panel.setLayout(new FlowLayout());
```

```
      panel.add(lblFirstName);
      panel.add(tfFirstName);
      panel.add(lblLastName);
      panel.add(tfLastName);
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setSize(300, 100);
      frame.getContentPane().add(panel, BorderLayout.CENTER);
      frame.setVisible(true);
  }
}
```

Output:

JTextArea:
- A **JTextArea** is a **multi-line text component** to display text or allow the user to enter text.
- A **JTextArea** will generate a **CaretListener** interface.
- The **JTextComponent** is a superclass of **JTextArea** that provides a common set of methods used by **JTextArea**.
- The important methods in the **JTextArea** class are **setText(), append(), setLineWrap(), setWrapStyleWord(), setCaretPosition()**, etc.

Example:
```
import java.awt.*;
import javax.swing.*;
import javax.swing.event.*;
public class JTextAreaTest {
  public static void main(String args[]) {
    JFrame frame = new JFrame("JTextArea Example");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    JTextArea textArea = new JTextArea();
    JScrollPane scrollPane = new JScrollPane(textArea);
    frame.add(scrollPane, BorderLayout.CENTER);
    CaretListener listener = new CaretListener() {
      public void caretUpdate(CaretEvent caretEvent) {
        System.out.println("Dot: "+ caretEvent.getDot());
```

```
      System.out.println("Mark: "+caretEvent.getMark());
    }
  };
  textArea.addCaretListener(listener);
  frame.setSize(250, 150);
  frame.setVisible(true);
 }
}
```

Output:
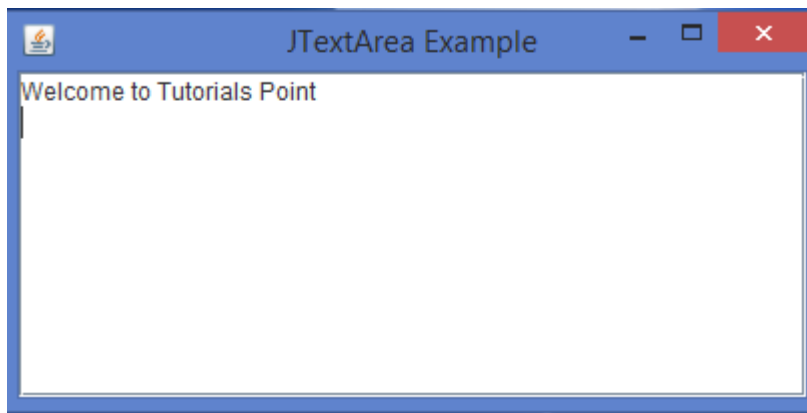
4.a) Describe the hierarchy of AWT user interface components.

Java AWT (Abstract Window Toolkit) is *an API to develop Graphical User Interface (GUI) or windows-based applications* in Java.
Java AWT components are platform-dependent i.e. components are displayed according to the view of operating system. AWT is heavy weight i.e. its components are using the resources of underlying operating system (OS).
The java.awt package provides classes for AWT API such as TextField, Label, TextArea, RadioButton, CheckBox, Choice, List etc.
The AWT tutorial will help the user to understand Java GUI programming in simple and easy steps.

Java AWT Hierarchy
The hierarchy of Java AWT classes are given below.
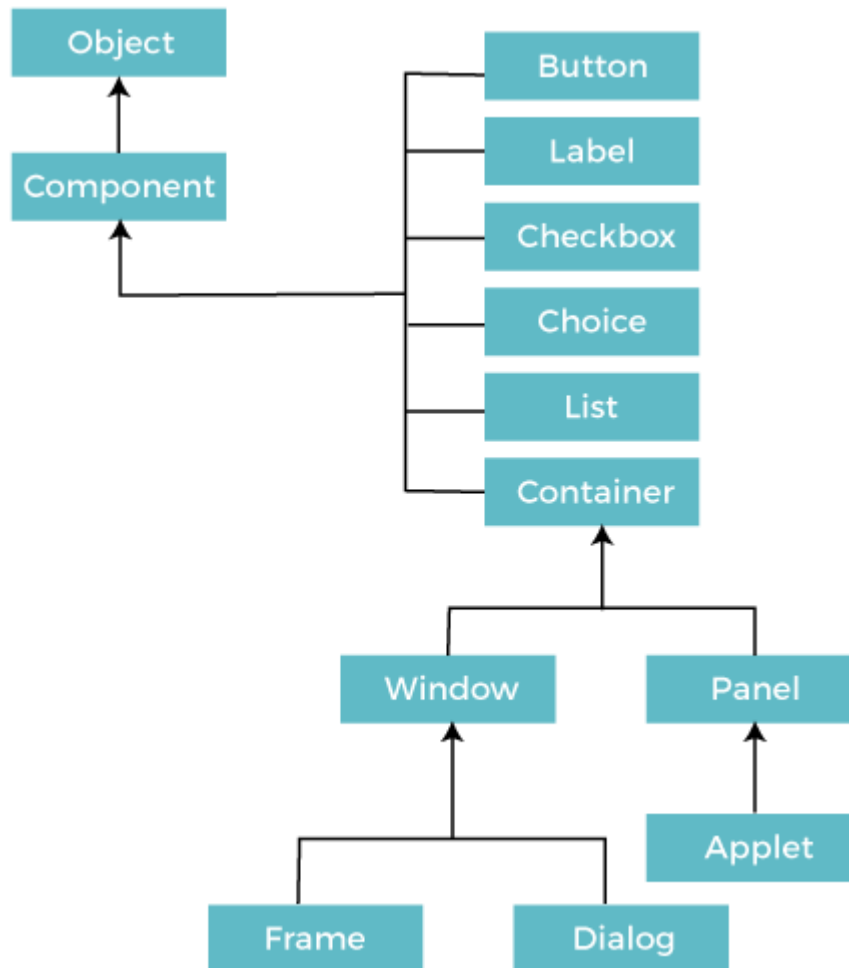
Components:

All the elements like the button, text fields, scroll bars, etc. are called components. In Java AWT, there are classes for each component as shown in above diagram. In order to place every component in a particular position on a screen, we need to add them to a container.
**Container:**

The Container is a component in AWT that can contain another components like buttons, textfields, labels etc. The classes that extends Container class are known as container such as **Frame, Dialog** and **Panel**.

Let's see a simple example of AWT where we are inheriting Frame class. Here, we are showing Button component on the Frame.

AWTExample1.java

```
1.  // importing Java AWT class
2.  import java.awt.*;
3.
4.  // extending Frame class to our class AWTExample1
5.  public class AWTExample1 extends Frame {
6.
7.    // initializing using constructor
8.    AWTExample1() {
9.
10.     // creating a button
11.     Button b = new Button("Click Me!!");
12.
13.     // setting button position on screen
14.     b.setBounds(30,100,80,30);
15.
16.     // adding button into frame
17.     add(b);
18.
19.     // frame size 300 width and 300 height
20.     setSize(300,300);
21.
22.     // setting the title of Frame
23.     setTitle("This is our basic AWT example");
24.
25.     // no layout manager
26.     setLayout(null);
27.
28.     // now frame will be visible, by default it is not visible
29.     setVisible(true);
30. }
31.
32. // main method
33. public static void main(String args[]) {
34.
35. // creating instance of Frame class
36. AWTExample1 f = new AWTExample1();
37.
38. }
39.
40. }
```

Output:
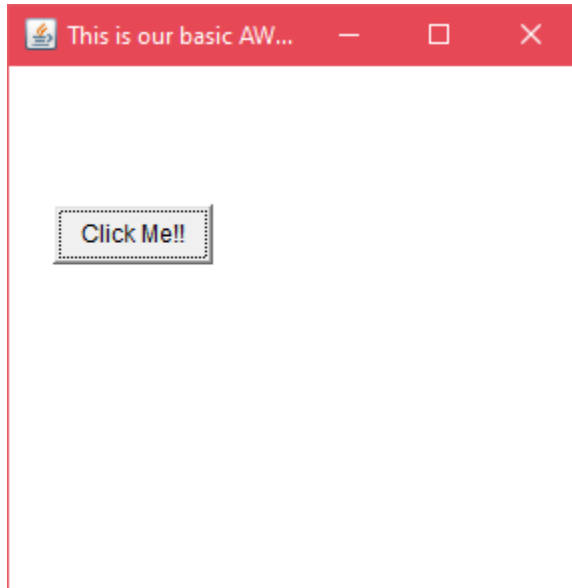
4.b) Explain the complete event handling mechanism in Java, covering event sources, listeners, event classes, and their interactions in detail.
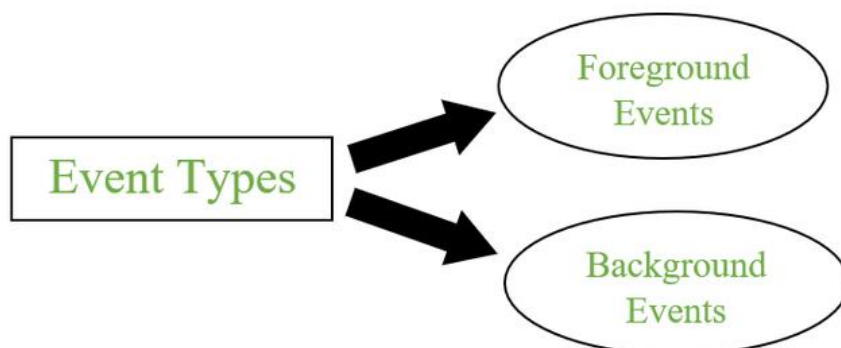
Event Handling in Java

An **event** can be defined as changing the state of an object or behavior by performing actions. Actions can be a button click, cursor movement, keypress through keyboard or page scrolling, etc.

The **java.awt.event** package can be used to provide various event classes.

Classification of Events
- Foreground Events
- Background Events

1. Foreground Events

Foreground events are the events that require user interaction to generate, i.e., foreground events are generated due to interaction by the user on components in Graphic User Interface (**GUI**). Interactions are nothing but clicking on a button, scrolling the scroll bar, cursor moments, etc.

2. Background Events

Events that don't require interactions of users to generate are known as background events. Examples of these events are operating system failures/interrupts, operation completion, etc.

Event Handling

It 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 Sources and Listeners.

Delegation Event Model
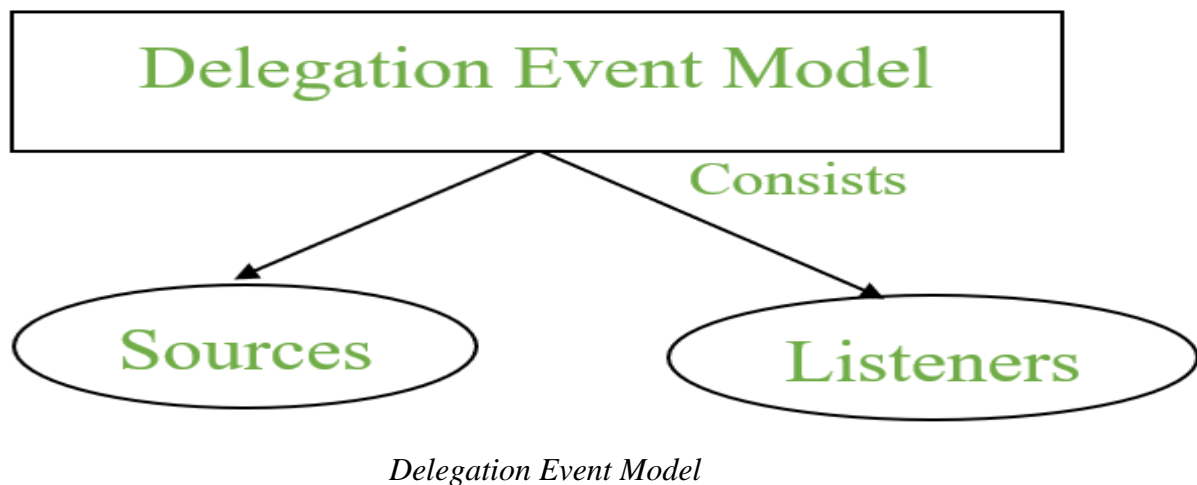
- **Source:** Events are generated from the source. There are various sources like buttons, checkboxes, list, menu-item, choice, scrollbar, text components, windows, etc., to generate events.
- **Listeners:** Listeners are used for handling the events generated from the source. Each of these listeners represents interfaces that are responsible for handling events.

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

Registering the Source with Listener:

Different Classes provide different registration methods.

Syntax:

addTypeListener()

where Type represents the type of event.

Example 1: For **KeyEvent** we use *addKeyListener()* to register.

Example 2: that For **ActionEvent** we use *addActionListener()* to register.

Event Classes in Java

Event Class	Listener Interface	Description
ActionEvent	ActionListener	An event that indicates that a component-defined action occurred like a button click or selecting an item from the menu-item list.
AdjustmentEvent	AdjustmentListener	The adjustment event is emitted by an Adjustable object like Scrollbar.
ComponentEvent	ComponentListener	An event that indicates that a component moved, the size changed or changed its visibility.
ContainerEvent	ContainerListener	When a component is added to a container (or) removed from it, then this event is generated by a container object.
FocusEvent	FocusListener	These are focus-related events, which include focus, focusin, focusout, and blur.
ItemEvent	ItemListener	An event that indicates whether an item was selected or not.
KeyEvent	KeyListener	An event that occurs due to a sequence of keypresses on the keyboard.
MouseEvent	MouseListener & MouseMotionListener	The events that occur due to the user interaction with the mouse (Pointing Device).
MouseWheelEvent	MouseWheelListener	An event that specifies that the mouse wheel was rotated in a component.
TextEvent	TextListener	An event that occurs when an object's text changes.

4.c) Discuss in detail 5 types of Layout Managers.
Ans**:** The Layout managers enable us to control the way in which visual components are arranged in the GUI forms by determining the size and position of components within the containers.

Types of Layout Manager:

There are 6 layout managers in Java
- **FlowLayout**: It arranges the components in a container like the words on a page. It fills the top line from **left to right and top to bottom**. The components are arranged in the order as they are added i.e. first components appears at top left, if the container is not wide enough to display all the components, it is wrapped around the line. Vertical and horizontal gap between components can be controlled. The components can be **left, center or right aligned.**
- **BorderLayout**: It arranges all the components along the edges or the middle of the container i.e. **top, bottom, right and left** edges of the area. The components added to the top or bottom gets its preferred height, but its width will be the width of the container and also the components added to the left or right gets its preferred width, but its height will be the remaining height of the container. The components added to the center gets neither its preferred height or width. It covers the remaining area of the container.
- **GridLayout**: It arranges all the components in a grid of **equally sized cells**, adding them from the **left to righ**t and **top to bottom**. Only one component can be placed in a cell and each region of the grid will have the same size. When the container is resized, all cells are automatically resized. The order of placing the components in a cell is determined as they were added.
- **GridBagLayout**: It is a powerful layout which arranges all the components in a grid of cells and maintains the aspect ration of the object whenever the container is resized. In this layout, cells may be different in size. It assigns a consistent horizontal and vertical gap among components. It allows us to specify a default alignment for components within the columns or rows.
- **BoxLayout**: It arranges multiple components in either **vertically or horizontally**, but not both. The components are arranged from **left to right or top to bottom**. If the components are aligned **horizontally**, the height of all components will be the same and equal to the largest sized components. If the components are aligned **vertically**, the width of all components will be the same and equal to the largest width components.
- **CardLayout**: It arranges two or more components having the same size. The components are **arranged in a deck**, where all the cards of the same size and the **only top card are visible at any time**. The first component added in the container will be kept at the top of the deck. The default gap at the left, right, top and bottom edges are zero and the card components are displayed either **horizontally or vertically.**

  Exmple:
```
import java.awt.*;
import javax.swing.*;
public class LayoutManagerTest extends JFrame {
   JPanel flowLayoutPanel1, flowLayoutPanel2, gridLayoutPanel1, gridLayoutPanel2,
gridLayoutPanel3;
   JButton one, two, three, four, five, six;
   JLabel bottom, lbl1, lbl2, lbl3;
   public LayoutManagerTest() {
     setTitle("LayoutManager Test");
     setLayout(new BorderLayout()); // Set BorderLayout for JFrame
     flowLayoutPanel1 = new JPanel();
```

```
    one = new JButton("One");
    two = new JButton("Two");
    three = new JButton("Three");
    flowLayoutPanel1.setLayout(new FlowLayout(FlowLayout.CENTER)); // Set
FlowLayout Manager
    flowLayoutPanel1.add(one);
    flowLayoutPanel1.add(two);
    flowLayoutPanel1.add(three);
    flowLayoutPanel2 = new JPanel();
    bottom = new JLabel("This is South");
    flowLayoutPanel2.setLayout (new FlowLayout(FlowLayout.CENTER)); // Set
FlowLayout Manager
    flowLayoutPanel2.add(bottom);
    gridLayoutPanel1 = new JPanel();
    gridLayoutPanel2 = new JPanel();
    gridLayoutPanel3 = new JPanel();
    lbl1 = new JLabel("One");
    lbl2 = new JLabel("Two");
    lbl3 = new JLabel("Three");
    four = new JButton("Four");
    five = new JButton("Five");
    six = new JButton("Six");
    gridLayoutPanel2.setLayout(new GridLayout(1, 3, 5, 5)); // Set GridLayout Manager
    gridLayoutPanel2.add(lbl1);
    gridLayoutPanel2.add(lbl2);
    gridLayoutPanel2.add(lbl3);
    gridLayoutPanel3.setLayout(new GridLayout(3, 1, 5, 5)); // Set GridLayout Manager
    gridLayoutPanel3.add(four);
    gridLayoutPanel3.add(five);
    gridLayoutPanel3.add(six);
    gridLayoutPanel1.setLayout(new GridLayout(2, 1)); // Set GridLayout Manager
    gridLayoutPanel1.add(gridLayoutPanel2);
    gridLayoutPanel1.add(gridLayoutPanel3);
    add(flowLayoutPanel1, BorderLayout.NORTH);
    add(flowLayoutPanel2, BorderLayout.SOUTH);
    add(gridLayoutPanel1, BorderLayout.CENTER);
    setSize(400, 325);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setLocationRelativeTo(null);
    setVisible(true);
  }
  public static void main(String args[]) {
    new LayoutManagerTest();
  }
}
```

Output:
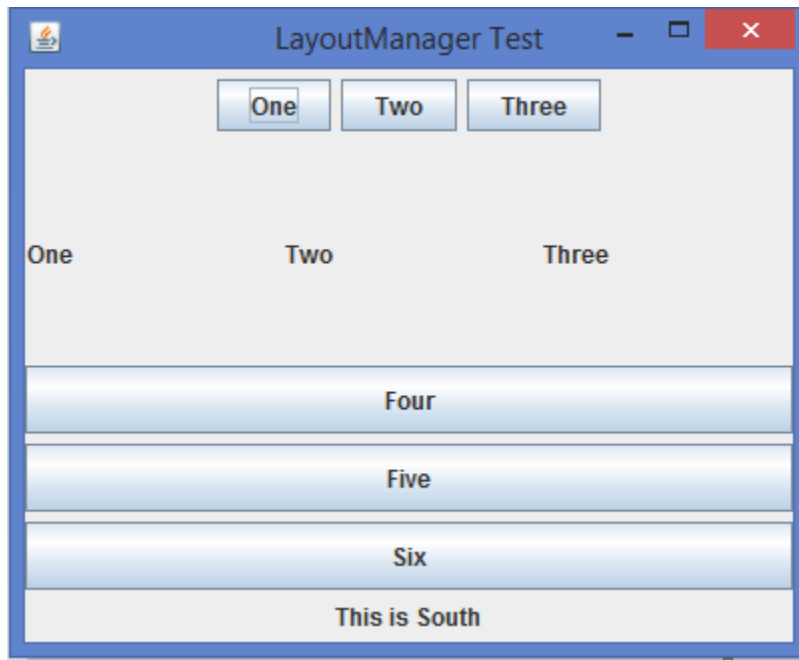
Q. No	Question (s)	Marks	BL	CO
	UNIT – V			
1	**a) Which container uses the boarder layout as their default layout.**	**1M**	L2	C214.6
	b) What are untrusted applets?	**1M**	L1	C214.5
	c)What type of program is embedded in webpage to generate the dynamic content	**1M**	L1	C214.6
	d)Which method is called when every time applet receives focus	**1M**	L1	C214.5
	e) What method is called to clear the screen and calls paint() method.	**1M**	L1	C214.5
2	**a) Write syntax and example of drawPolygon(), drawRect(), drawArc()**	**3M**	L1	C214.5
	b) What is an event and what are the modules available for event handling	**3M**	L1	C214.6
	c) Difference between component and container	**3M**	L3	C214.6
	d)What are the restrictions imposed on java applets	**3M**	L1	C214.5
	e) What is the source and listener	**3M**	L1	C214.6
3	**a) Differentiate between Applet and Application**	**5M**	L3	C214.5
	b) Explain the following methods of applet class, drawRect(), drawPolygon(), drawArc(), drawRoundRect().	**5M**	L2	C214.5
	c)Provide the syntax of the following methods of graphics class and explain with program. a) drawOval() b)drawaPolygon()	**5M**	L2	C214.5

	d) **Define applet and write a program to display message "Welcome to Java"**	**5M**	L1	C214.5
	e) **Explain any 4 applet tag with proper explanation.**	**5M**	L1	C214.5
4	a) **Explain in detail MVC architecture.**	**10M**	L1	C214.6
	b) **Explain applet lifecycle with diagram.**	**10M**	L1	C214.6
	c) **Explain different swing components.**	**10M**	L1	C214.6

1. **Which container uses the boarder layout as their defaultlayout.**
Ans:
In Java's Swing library, the `JFrame` uses the BorderLayout as its default layout manager. When you add components to a `JFrame`, they are positioned in one of the five regions: North, South, East, West, or Center. If you don't specify a layout manager, components will be arranged according to this layout.

2. **What are untrusted applets?**

Ans: Ans: Untrusted applets are Java programs that run in a web browser and have limited permissions due to security concerns. Since they can be downloaded from the internet, they are considered "untrusted" because they may pose risks to the user's system.

Key characteristics include:

1. **Sandboxing**: They run in a restricted environment (sandbox) that prevents them from performing potentially harmful operations, such as accessing the file system or network resources outside of their host server.
2. **Limited Access**: Untrusted applets cannot access sensitive information on the user's machine or make system-level changes.
3. **Security Manager**: The Java security manager enforces these restrictions to ensure the applet behaves safely.
4. **Use Cases**: They are often used for interactive content in web pages, like games or simple animations, without compromising user security.

3. **What type of program is embedded in webpage to generate the dynamic content.**
   Ans:
   The type of program typically embedded in a webpage to generate dynamic content is often referred to as a **server-side script**. Common technologies for this purpose include:

1. **PHP**: Widely used for web development and can generate HTML dynamically.
2. **Python** (with frameworks like Django or Flask): Allows for creating dynamic web applications.
3. **Ruby on Rails**: A framework for building dynamic web applications in Ruby.
4. **Node.js**: JavaScript runtime that can create dynamic web servers and applications.

4. **Which method is called when every time applet receives focus?**
   Ans: In Java applets, the method that is called every time the applet receives focus is the

focusGained(FocusEvent e) method. This method is part of the FocusListener interface. To handle focus events, you would implement the FocusListener interface and override the focusGained method:

import java.applet.Applet;
import java.awt.event.FocusEvent;
import java.awt.event.FocusListener;

public class MyApplet extends Applet implements FocusListener {

   public void init() {
     addFocusListener(this);
   }

   @Override
   public void focusGained(FocusEvent e) {
     // Code to execute when the applet gains focus
   }

   @Override
   public void focusLost(FocusEvent e) {
     // Code to execute when the applet loses focus
   }
}

5. **What method is called to clear the screen and calls paint()method.**
Ans:
In Java applets (and also in Swing applications), the method used to clear the screen and call the paint() method is typically repaint().When you call repaint(), it schedules a call to the update(Graphics g) method, which in turn calls the paint(Graphics g) method. To effectively clear the screen and redraw your content, you can override the update() method like this:
import java.applet.Applet;
import java.awt.Graphics;
public class MyApplet extends Applet {

   @Override
   public void paint(Graphics g) {
     // Code to draw on the applet
   }

   @Override
   public void update(Graphics g) {
     // Clear the screen by filling it with a background color
     g.clearRect(0, 0, getWidth(), getHeight());
     // Call the paint method to redraw the content
     paint(g);
   }

```
    public void clearScreen() {
        // This will clear the screen and trigger the repaint
        repaint();
    }
}
```

6. **Write syntax and example of drawPolygon(), drawRect(),drawArc()?**
   Ans:
   In Java, you can use the `Graphics` class to draw shapes like polygons, rectangles, and arcs. Below is the syntax and an example for each method: `drawPolygon()`, `drawRect()`, and `drawArc()`.

   `drawPolygon()`: syntax: void drawPolygon(int[] xPoints, int[] yPoints, int nPoints)

   `drawRect()`: syntax : void drawRect(int x, int y, int width, int height)

   `drawArc()`:syntax: void drawArc(int x, int y, int width, int height, int startAngle, int arcAngle).

7. **What is an event and what are the modules available for eventhandling.**
   Events are signals fired inside the browser window that notify of changes in the browser or operating system environment. Programmers can create event handler code that will run when an event fires, allowing web pages to respond appropriately to change

8. **Difference between component and container.**
   A **component:** is a reusable, self-contained unit of functionality or UI in a software application. Components typically encapsulate both logic and presentation and can be as simple as a button or as complex as an entire form or widget. They are the building blocks of an interface.

   A **container:** is a special type of component whose primary role is to **hold** or **manage other components**. Containers are responsible for organizing and structuring the layout and behavior of child components. While containers can have some logic or presentation of their own, their main job is to manage how components inside them are arranged and how they interact.

9. **What are the restrictions imposed on java applets.**
   Java applets were small applications that ran within a web browser, but over time they became obsolete due to security concerns and the rise of more modern web technologies like HTML5, JavaScript, and CSS. Despite this, it's useful to know the **restrictions** that were imposed on Java applets when they were still in use, primarily for security reasons.

- **Sandbox Security Model**: Applets ran in a *sandbox*, which was a restricted execution environment designed to prevent potentially malicious activities. The sandbox placed strict limitations on what an applet could do.

- **File System Access**:

  - **Restricted**: Applets could not read from or write to the local file system on the user's machine. This was to prevent malicious applets from accessing sensitive files or writing harmful content.

- **Exception**: Digitally signed applets (that the user explicitly trusted) could request permission to access the local file system.

- **Network Access**:

  - **Restricted**: An unsigned applet could only communicate with the server from which it was downloaded. It couldn't make network connections to other servers or hosts, preventing the applet from being used to launch attacks on other machines.
  - **Exception**: Signed applets could request more lenient network permissions.

  10. **What is the source and listener?**
      Ans:
      event-driven programming, the terms **source** and **listener** play crucial roles in handling events (such as user actions or system events). This model is used to decouple components in a system, allowing them to respond to events without needing direct knowledge of each other.

Examples of Sources:

- A button in a GUI is a source when clicked by the user.
- A text field is a source when a user types something in it.
- A server can be a source when a message is received.

A **listener** is an object that **responds to events** generated by an event source. It "listens" for specific types of events and defines how to handle them when they occur. The listener registers itself with a source, and when the event happens, the source notifies the listener by calling specific methods on it.

- **Examples of Listeners**:
  - A mouse click listener that responds when a button is clicked.
  - A key listener that responds when a key is pressed on the keyboard.
  - A network event listener that responds when a message arrives from the network.

  11. **Differentiate between Applet and Application.**

Java Application Vs. Java Applet

Parameters	Java Application	Java Applet
Meaning	A Java Application also known as application program is a type of program that independently executes on the computer.	The Java applet works on the client side, and runs on the browser and makes use of another application program so that we can execute it.
Requirement of	Its execution starts	It does not require the use of any main() method.

main( ) method	with the main( ) method only. The use of the main( ) is mandatory.	Java applet initializes through init( ) method.
Execution	It cannot run independently, but requires JRE to run.	It cannot start independently but requires APIs for use (Example. APIs like Web API).
Installation	We need to install the Java application first and obviously on the local computer.	Java applet does not need to be pre-installed.
Connectivity with server	It is possible to establish connections with other servers.	It cannot establish connection to other servers.
Operation	It performs read and write tasks on a variety of files located on a local computer.	It cannot run the applications on any local computer.
File access	It can easily access a file or data available on a computer system or device.	It cannot access the file or data found on any local system or computer.
Security	Java applications are pretty trusted, and thus, come with no security concerns.	Java applets are less reliable. So, they need to be safe.

12. **Explain the following methods of applet class, drawRect(), drawPolygon(), drawArc(), drawRoundRect().**

The methods `drawRect()`, `drawPolygon()`, `drawArc()`, and `drawRoundRect()` are part of the **Graphics class** in Java, which is commonly used in applets and other graphical user interfaces (GUI) to draw shapes on the screen. These methods are typically called within the `paint()` method of an applet or component, allowing you to draw different shapes. Here's a detailed explanation of each method:

`drawRect(int x, int y, int width, int height):`

This method is used to draw a **rectangle**. The rectangle is defined by its top-left corner (specified by `x` and `y`) and its dimensions (`width` and `height`).

```
drawPolygon(int[] xPoints, int[] yPoints, int nPoints):
```

This method is used to draw a **polygon**, which is a closed shape consisting of connected line segments. The vertices of the polygon are specified by two arrays of coordinates: `xPoints` (for the x-coordinates) and `yPoints` (for the y-coordinates

```
drawArc(int x, int y, int width, int height, int startAngle, int arcAngle):
```

This method is used to draw an **arc**. The arc is part of an oval or circle that fits within the rectangle defined by the `x`, `y`, `width`, and `height` parameters. The arc starts at `startAngle` and spans `arcAngle` degrees.

```
drawRoundRect(int x, int y, int width, int height, int arcWidth, int
arcHeight):
```

This method is used to draw a **rounded rectangle**. The rectangle has rounded corners defined by `arcWidth` and `arcHeight`.

Method	Description
`drawRect(x, y, width, height)`	Draws a rectangle using the specified x, y, width, and height.
`drawPolygon(xPoints, yPoints, nPoints)`	Draws a polygon by connecting the points specified by the xPoints and yPoints arrays.
`drawArc(x, y, width, height, startAngle, arcAngle)`	Draws an arc inside a bounding rectangle with a specified start angle and angular extent.
`drawRoundRect(x, y, width, height, arcWidth, arcHeight)`	Draws a rectangle with rounded corners using the specified arc dimensions for rounding.

13. Provide the syntax of the following methods of graphics class and explain with program. a) drawOval() b)drawaPolygon()

the `Graphics` class is used to perform various drawing operations, such as drawing shapes, text, or images. Two important methods for drawing shapes are `drawOval()` and `drawPolygon()`. Below, I will explain both methods and provide example programs to demonstrate their usage.

The `drawOval()` method draws an outline of an oval (or ellipse) that fits within the bounding rectangle specified by the `(x, y)` coordinates and the specified `width` and `height`.

If the `width` and `height` are equal, the oval will be a circle.

import java.applet.Applet;

import java.awt.Graphics;

```
/*

  <applet code="DrawOvalApplet.class" width="300" height="300">

  </applet>

*/
```

```
public class DrawOvalApplet extends Applet {

  public void paint(Graphics g) {

    // Drawing an oval inside a rectangle with top-left corner at (50, 50), width 150, height 100

    g.drawOval(50, 50, 150, 100);


    // Drawing a circle by specifying equal width and height

    g.drawOval(100, 200, 100, 100); // Circle with center (100,200) and radius 50

  }

}
```

The `drawPolygon()` method draws a polygon by connecting a series of points defined by two arrays: one for the x-coordinates and one for the y-coordinates. The `nPoints` parameter specifies how many vertices (points) the polygon has. The polygon is automatically closed by connecting the last point to the first point.

```
import java.applet.Applet;

import java.awt.Graphics;


/*

  <applet code="DrawPolygonApplet.class" width="300" height="300">

  </applet>

*/
```

```java
public class DrawPolygonApplet extends Applet {

    public void paint(Graphics g) {

        // Defining x and y coordinates for a triangle

        int[] xPoints = {50, 150, 100};

        int[] yPoints = {100, 100, 50};

        int nPoints = 3;


        // Drawing the triangle

        g.drawPolygon(xPoints, yPoints, nPoints);


        // Defining x and y coordinates for a pentagon

        int[] xPentagon = {200, 240, 280, 260, 220};

        int[] yPentagon = {150, 100, 150, 200, 200};

        int pentagonPoints = 5;


        // Drawing the pentagon

        g.drawPolygon(xPentagon, yPentagon, pentagonPoints);

    }

}
```

14. Define applet and write a program to display message "Welcome to Java"

Java applet is a small dynamic Java program that can be transferred via the Internet and run by a Java-compatible Web browser. The main difference between Java-based applications and applets is that applets are typically executed in an appletviewer or Javacompatible Web browser. All applets

import the java.awt package. /**/
import java.applet.*;
 import java.awt.*;
public class WelcomeJava extends Applet
{
public void paint(Graphics g)
 {
 g.drawString("Welcome to java",25,50);
 }
 }

15. **Explain any 4 applet tag with proper explanation**

The `<applet>` tag is used in HTML to embed a Java applet in a webpage. Even though applets are considered obsolete today due to browser security concerns and modern alternatives like HTML5 and JavaScript, understanding the tag and its attributes can still be valuable historically or when dealing with legacy systems.

Here are 4 commonly used attributes of the `<applet>` tag:

code Attribute:

- **Description**: The `code` attribute specifies the name of the Java class file that contains the applet's compiled code (the `.class` file). This class file should be available on the web server hosting the webpage, or it can be referenced via a URL.

width and height Attributes:

- **Description**: These attributes specify the dimensions of the applet's display area on the webpage. The width and height are given in pixels.

codebase Attribute:

- **Description**: The `codebase` attribute specifies the URL or directory path from which the applet's class files will be loaded. It allows you to place the applet's `.class` file on a different server or directory from the HTML file.

param Tag:

- **Description**: The `<param>` tag is used to pass additional parameters to the applet. These parameters can be accessed within the applet code using `getParameter()` method.

  <html>
    <body>
      <applet codebase="http://www.example.com/applets/" code="MyApplet.class" width="300" height="150">
        <param name="bgColor" value="yellow">

```
         <param name="message" value="Hello from Applet">
      </applet>
   </body>
</html>
```

16. Explain in detail MVC architecture.
Ans:
The model designs based on the MVC architecture follow MVC design pattern. The application logic is separated from the user interface while designing the software using model designs.The MVC pattern architecture consists of three layers

o **Model:** It represents the business layer of application. It is an object to carry the data that can also contain the logic to update controller if data is changed.
o **View:** It represents the presentation layer of application. It is used to visualize the data that the model contains.
o **Controller:** It works on both the model and view. It is used to manage the flow of application, i.e. data flow in the model object and to update the view whenever data is changed.

In Java Programming, the Model contains the simple Java classes, the View used to display the data and the Controller contains the servlets. Due to this separation the user requests are processed as follows:
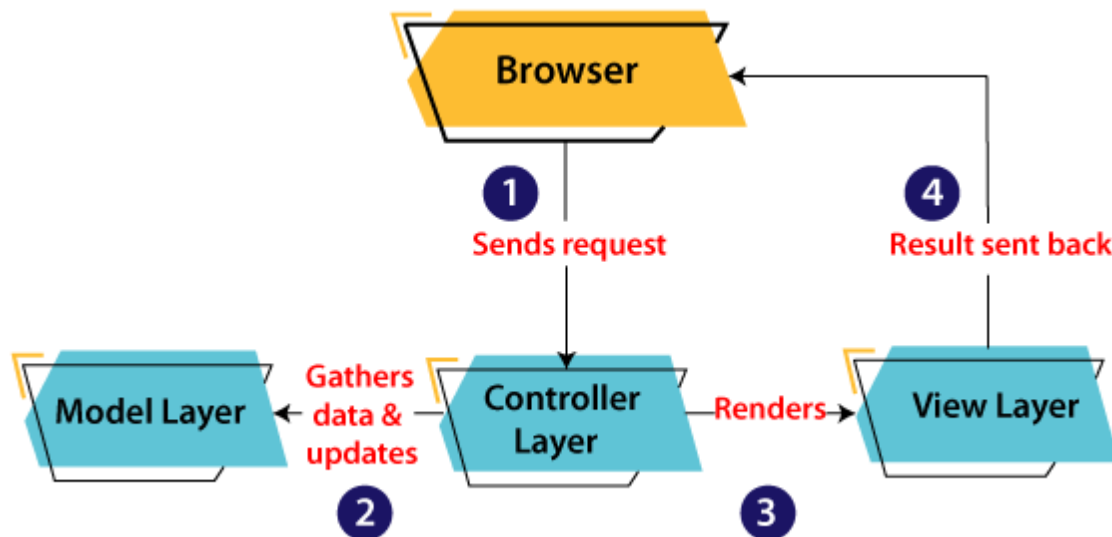

1. A client (browser) sends a request to the controller on the server side, for a page.
2. The controller then calls the model. It gathers the requested data.
3. Then the controller transfers the data retrieved to the view layer.
4. Now the result is sent back to the browser (client) by the view.

17. **Explain applet lifecycle with diagram.**

an <u>applet</u> is a special type of program embedded in the web page to generate dynamic content. Applet is a class in Java.

The applet life cycle can be defined as the process of how the object is created, started, stopped, and destroyed during the entire execution of its application. It basically has five core methods namely init(), start(), stop(), paint() and destroy().These methods are invoked by the browser to execute.

- Along with the browser, the applet also works on the client side, thus having less processing time.

- Methods of Applet Life Cycle
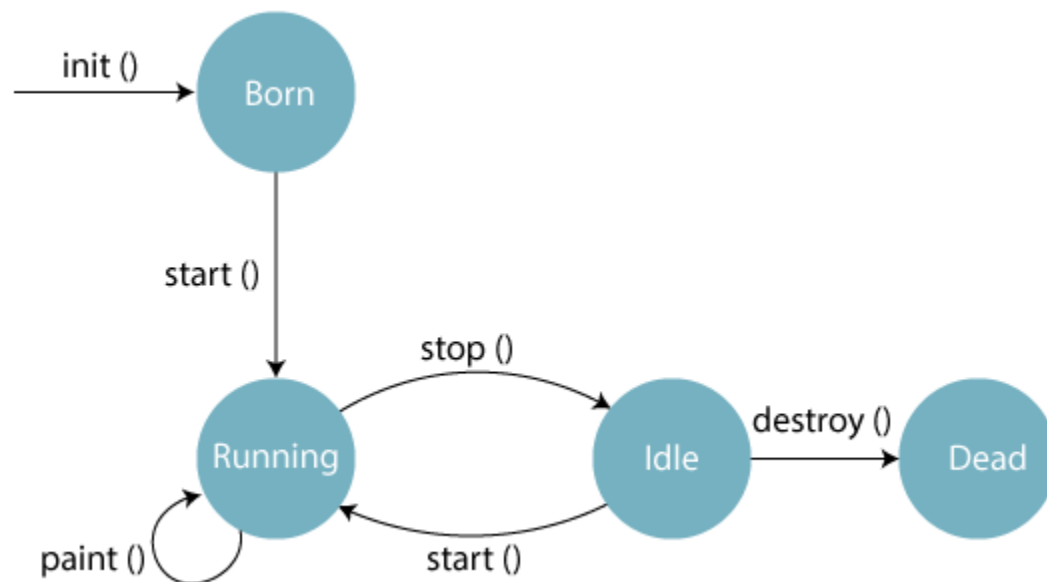
-
- There are five methods of an applet life cycle, and they are:
- **init():** The init() method is the first method to run that initializes the applet. It can be invoked only once at the time of initialization. The web browser creates the initialized objects, i.e., the web browser (after checking the security settings) runs the init() method within the applet.
- **start():** The start() method contains the actual code of the applet and starts the applet. It is invoked immediately after the init() method is invoked. Every time the browser is loaded or refreshed, the start() method is invoked. It is also invoked whenever the applet is maximized, restored, or moving from one tab to another in the browser. It is in an inactive state until the init() method is invoked.
- **stop():** The stop() method stops the execution of the applet. The stop () method is invoked whenever the applet is stopped, minimized, or moving from one tab to another in the browser, the stop() method is invoked. When we go back to that page, the start() method is invoked again.
- **destroy():** The destroy() method destroys the applet after its work is done. It is invoked when the applet window is closed or when the tab containing the webpage is closed. It removes the applet object from memory and is executed only once. We cannot start the applet once it is destroyed.
- **paint():** The paint() method belongs to the Graphics class in Java. It is used to draw shapes like circle, square, trapezium, etc., in the applet. It is executed after the start() method and when the browser or applet windows are resized.

18. **Explain different swing components.**

JButton : The JButton class is used to create a labeled button that has platform independent implementation.
  The application result in some action when the button is pushed.
It inherits AbstractButton class.

JLabel The object of JLabel class is a component for placing text in a container.
 It is used to display a single line of read only text.
 The text can be changed by an application but a user cannot edit it directly.
It inherits JComponent class.

JTextField The object of a JTextField class is a text component that allows the editing of a single line text.
It inherits JTextComponent class

JTextArea The object of a JTextArea class is a multiline region that displays text. It allows the editing of multiple line text. It inherits JTextComponent class

JCheckBox The JCheckBox class is used to create a checkbox. It is used to turn an option on (true) or off (false).
 Clicking on a CheckBox changes its state from "on" to "off" or from "off" to "on ".
It inherits JToggleButton class
JRadioButton The JRadioButton class is used to create a 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

JComboBox The object of Choice class is used to show popup menu of choices.
 Choice selected by user is shown on the top of a menu
 It inherits JComponent class
JTree :The JTree class is used to display the tree structured data or hierarchical data.

  JTree is a complex component. It has a 'root node' at the top most which is a parent for all nodes in
 the tree. It inherits JComponent class

JTabbedPane The JTabbedPane class is used to switch between a group of components by clicking on a tab with a given title or icon. It inherits JComponent class

ScrollPane A JscrollPane is used to make scrollable view of a component. When screen size is limited, we use a scroll pane to display a large component or a component whose size can change dynamically.