**CS3391 OBJECT ORIENTED PROGRAMMING**

**PART-C**

**UNIT I INTRODUCTION TO OOP AND JAVA**

**1. (i) Explain the characteristics of OOPs (Nov/Dec 2018)**

**Introduction:**

Object-Oriented Programming (OOP) is a programming paradigm centered around the concept of "objects" which contain data in the form of fields and code in the form of procedures or methods.

**Characteristics of OOP:**

1. **Class and Object:**
   * **Class:** A blueprint for creating objects.
   * **Object:** A real-world entity created from a class.
   * *Example:*

java

CopyEdit

class Student {

String name;

int age;

}

Student s1 = new Student(); // s1 is an object

1. **Encapsulation:**
   * Binding data and methods that manipulate the data into a single unit (class).
   * Helps in hiding internal implementation.
   * *Example:*

java

CopyEdit

class Account {

private double balance;

public void deposit(double amount) {

balance += amount;

}

}

1. **Abstraction:**
   * Hides complexity by showing only essential features.
   * Achieved using abstract classes and interfaces.
   * *Example:*

java

CopyEdit

abstract class Shape {

abstract void draw();

}

1. **Inheritance:**
   * Enables a new class to inherit properties of an existing class.
   * Promotes code reusability.
   * *Example:*

java

CopyEdit

class Animal {

void sound() { System.out.println("Animal Sound"); }

}

class Dog extends Animal {

void sound() { System.out.println("Bark"); }

}

1. **Polymorphism:**
   * Same function behaves differently based on context.
   * Two types: Compile-time (method overloading), Run-time (method overriding).
   * *Example:*

java

CopyEdit

class Math {

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

double add(double a, double b) { return a + b; }

}

1. **Message Passing:**
   * Objects communicate by sending and receiving information (messages).
   * *Example:*

java

CopyEdit

obj.display(); // message sent to object

1. **Modularity:**
   * Breaking down the program into smaller parts (modules or classes).
   * Enhances maintainability and reusability.
2. **Dynamic Binding:**
   * Decision about method to be invoked is made at runtime.
   * Achieved through method overriding and inheritance.

**1. (ii) Explain the features and characteristics of JAVA (Nov/Dec 2019)**

**Introduction:**

Java is a high-level, platform-independent, object-oriented programming language developed by Sun Microsystems.

**Features and Characteristics of Java:**

1. **Simple:**
   * Easy to learn; syntax similar to C/C++ but with simplified features.
2. **Object-Oriented:**
   * Follows OOP concepts: Class, Object, Inheritance, Polymorphism, Abstraction, and Encapsulation.
3. **Platform Independent:**
   * Java code is compiled into bytecode (.class file) that runs on any system with JVM.
4. **Secure:**
   * Features like no pointers, security manager, and bytecode verification ensure strong security.
5. **Robust:**
   * Strong memory management, exception handling, and garbage collection.
6. **Multithreaded:**
   * Supports multithreaded programming, allowing multiple tasks to run concurrently.
7. **Architecture-Neutral:**
   * Bytecode can run on any processor, provided the JVM is available.
8. **High Performance:**
   * Java performance is improved using JIT (Just-In-Time) compilers.
9. **Distributed:**
   * Supports networking and RMI (Remote Method Invocation).
10. **Dynamic:**

* Java can dynamically load classes during runtime.

1. **Portable:**

* Programs written in Java can be run on any machine with JVM.

1. **Interpreted and Compiled:**

* Java code is compiled into bytecode and then interpreted/executed by the JVM.

**2. (i) Describe the typical Java program structure.**

**Introduction:**

A Java program follows a standard structure to ensure readability, maintainability, and successful execution.

**Structure of a Typical Java Program:**

java

CopyEdit

// 1. Package Declaration (optional)

package mypackage;

// 2. Import Statements

import java.util.Scanner;

// 3. Class Definition

public class HelloWorld {

// 4. Main Method - Entry point of the program

public static void main(String[] args) {

// 5. Variable Declaration

String name = "Java";

// 6. Method Invocation / Logic

System.out.println("Hello, " + name);

}

}

**Explanation of Components:**

1. **Package Declaration:**
   * Organizes classes into namespaces.
   * Optional but useful in large projects.
2. **Import Statements:**
   * Used to include Java built-in packages and classes.
   * Example: import java.util.Scanner;
3. **Class Definition:**
   * Everything in Java is part of a class.
   * The filename should match the class name for public classes.
4. **Main Method:**
   * Acts as the entry point of any Java program.
   * public static void main(String[] args)
5. **Variables and Statements:**
   * Actual code logic, declaration of variables, method calls, loops, etc.
6. **Methods (Optional):**
   * User-defined reusable blocks of code.

**2. (ii) Explain the general Java program compilation and execution.**

**Step-by-Step Compilation and Execution:**

1. **Write the Java Code:**
   * Use any editor (Notepad, VS Code, Eclipse, etc.)
   * Save the file as ClassName.java
2. **Compile the Program:**
   * Use the Java Compiler:

nginx

CopyEdit

javac HelloWorld.java

* + Output: HelloWorld.class (Bytecode)

1. **Execute the Program:**
   * Use the Java Interpreter:

nginx

CopyEdit

java HelloWorld

* + Output: Runs the program and displays result

**Diagram:**

rust

CopyEdit

[HelloWorld.java] --> javac --> [HelloWorld.class] --> java --> [Output]

**Bytecode and JVM:**

* Bytecode: Intermediate code, not machine-specific.
* JVM: Java Virtual Machine interprets and runs bytecode.

**3. What are the different data types in JAVA? Explain each of them with example.**

**Java Data Types:**

Java supports two types of data types:

**1. Primitive Data Types (8 types)**

| **Type** | **Size** | **Example** |
| --- | --- | --- |
| byte | 1 byte | byte b = 100; |
| short | 2 bytes | short s = 10000; |
| int | 4 bytes | int i = 12345; |
| long | 8 bytes | long l = 999999L; |
| float | 4 bytes | float f = 12.34f; |
| double | 8 bytes | double d = 123.456; |
| char | 2 bytes | char c = 'A'; |
| boolean | 1 bit | boolean flag = true; |

**2. Non-Primitive Data Types**

* **String:**

java

CopyEdit

String name = "Java";

* **Arrays:**

java

CopyEdit

int[] numbers = {1, 2, 3};

* **Class and Objects:**

java

CopyEdit

class Student {

String name;

}

Student s = new Student();

**4. How to pass and return the objects to and from the method?**

**Passing Object to Method:**

java

CopyEdit

class Student {

int rollNo;

String name;

void display(Student s) {

System.out.println("Roll No: " + s.rollNo);

System.out.println("Name: " + s.name);

}

}

* In the above example, Student object is passed to display() method.

**Returning Object from Method:**

java

CopyEdit

class Student {

int rollNo;

String name;

Student getStudent() {

Student s = new Student();

s.rollNo = 101;

s.name = "Rahul";

return s;

}

}

* The method getStudent() returns an object of type Student.

**Example with both:**

java

CopyEdit

public class Test {

static Student createStudent() {

Student s = new Student();

s.rollNo = 1;

s.name = "Raj";

return s;

}

static void printStudent(Student s) {

System.out.println("Roll No: " + s.rollNo + ", Name: " + s.name);

}

public static void main(String[] args) {

Student myStudent = createStudent();

printStudent(myStudent);

}

}

**5. Discuss in detail the access specifiers available in Java.**

**Access Specifiers in Java:**

Java provides **four types of access specifiers** that control the visibility of classes, methods, and variables.

| **Modifier** | **Class** | **Package** | **Subclass** | **World** |
| --- | --- | --- | --- | --- |
| **public** | Yes | Yes | Yes | Yes |
| **protected** | Yes | Yes | Yes | No |
| **default** | Yes | Yes | No | No |
| **private** | Yes | No | No | No |

**1. public:**

* Accessible from **anywhere**.
* *Example:*

java

CopyEdit

public class MyClass {

public int number;

}

**2. private:**

* Accessible **only within the same class**.
* *Used for data hiding.*
* *Example:*

java

CopyEdit

class MyClass {

private int data = 10;

}

**3. protected:**

* Accessible **within the same package** and **by subclasses** in other packages.
* *Used in inheritance.*
* *Example:*

java

CopyEdit

class Animal {

protected void sound() {

System.out.println("Animal Sound");

}

}

**4. Default (Package-Private):**

* When no access specifier is mentioned.
* Accessible **only within the same package**.
* *Example:*

java

CopyEdit

class MyClass {

int data = 5; // default access

}

**UNIT II INHERITANCE, PACKAGES AND INTERFACES**

## ****1. Define Inheritance. With diagrammatic illustration and Java programs illustrate the different types of inheritance.****

(Nov./Dec. 2018, Nov/Dec 2019)

### ****Definition:****

**Inheritance** is a mechanism in Java by which one class can acquire the properties (fields) and behaviors (methods) of another class. It promotes **code reusability** and supports **hierarchical classification**.

### ****Syntax:****

java

CopyEdit

class Parent {

// parent properties

}

class Child extends Parent {

// child properties

}

### ****Types of Inheritance in Java:****

#### ****1. Single Inheritance****

java

CopyEdit

class Animal {

void sound() { System.out.println("Animal makes sound"); }

}

class Dog extends Animal {

void bark() { System.out.println("Dog barks"); }

}

public class Test {

public static void main(String[] args) {

Dog d = new Dog();

d.sound();

d.bark();

}

}

**Diagram:**

nginx

CopyEdit

Animal → Dog

#### ****2. Multilevel Inheritance****

java

CopyEdit

class Animal {

void eat() { System.out.println("Eating..."); }

}

class Dog extends Animal {

void bark() { System.out.println("Barking..."); }

}

class Puppy extends Dog {

void weep() { System.out.println("Weeping..."); }

}

public class Test {

public static void main(String[] args) {

Puppy p = new Puppy();

p.eat(); p.bark(); p.weep();

}

}

**Diagram:**

nginx

CopyEdit

Animal → Dog → Puppy

#### ****3. Hierarchical Inheritance****

java

CopyEdit

class Animal {

void eat() { System.out.println("Eating..."); }

}

class Dog extends Animal {

void bark() { System.out.println("Barking..."); }

}

class Cat extends Animal {

void meow() { System.out.println("Meowing..."); }

}

public class Test {

public static void main(String[] args) {

Dog d = new Dog();

Cat c = new Cat();

d.eat(); d.bark();

c.eat(); c.meow();

}

}

**Diagram:**

markdown

CopyEdit

Animal

/ \

Dog Cat

Note: Java **does not support multiple inheritance** using classes to avoid ambiguity (diamond problem). It can be achieved using **interfaces**.

## ****2. What is an interface? With an example explain how to define and implement interface.****

(Nov/Dec 2020, Apr/May 2021)

### ****Definition:****

An **interface** in Java is a blueprint of a class. It contains **abstract methods** and **constants**. Interfaces support **multiple inheritance** and **loose coupling**.

### ****Syntax:****

java

CopyEdit

interface MyInterface {

void display(); // abstract method

}

### ****Implementation:****

java

CopyEdit

interface Drawable {

void draw(); // abstract method

}

class Circle implements Drawable {

public void draw() {

System.out.println("Drawing Circle");

}

}

public class Test {

public static void main(String[] args) {

Drawable d = new Circle();

d.draw();

}

}

### ****Key Points:****

* All methods in interface are implicitly **public and abstract**.
* Fields are **public, static, and final**.
* A class uses implements to use an interface.
* A class can implement **multiple interfaces**.

## ****3. Differentiate method overloading and method overriding. Explain both with example.****

| **Feature** | **Method Overloading** | **Method Overriding** |
| --- | --- | --- |
| Definition | Same method name, different parameters | Subclass redefines parent method |
| Compile-time/runtime | Compile-time polymorphism | Runtime polymorphism |
| Inheritance needed? | No | Yes |
| Access modifier | Can be any | Must be same or more accessible |
| Return type | Can be different | Must be same or covariant |

### ****Example of Method Overloading:****

java

CopyEdit

class Adder {

int add(int a, int b) {

return a + b;

}

double add(double a, double b) {

return a + b;

}

}

### ****Example of Method Overriding:****

java

CopyEdit

class Animal {

void sound() {

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

}

}

class Dog extends Animal {

void sound() {

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

}

}

**UNIT III EXCEPTION HANDLING AND MULTITHREADING**

**1. Explain in detail the important methods of Java Exception Class**

**Introduction:**

In Java, all exceptions are derived from the class Throwable, which has two main subclasses:

* Exception
* Error

The Exception class is used for handling **run-time exceptions**. Java provides several important methods in the Throwable class (and thus available in Exception) to work with exception objects.

**Important Methods of Exception Class:**

| **Method** | **Description** |
| --- | --- |
| String getMessage() | Returns a detailed message about the exception. |
| String toString() | Returns a string representation of the exception. |
| void printStackTrace() | Prints the stack trace to the standard error stream. |
| StackTraceElement[] getStackTrace() | Returns an array containing the stack trace elements. |
| Throwable getCause() | Returns the cause of the exception. |
| void initCause(Throwable cause) | Initializes the cause of the exception. |
| Throwable fillInStackTrace() | Records the stack trace and returns the throwable object. |

**Example Program:**

java

CopyEdit

public class ExceptionMethodsDemo {

public static void main(String[] args) {

try {

int a = 10 / 0;

} catch (ArithmeticException e) {

System.out.println("getMessage(): " + e.getMessage());

System.out.println("toString(): " + e.toString());

System.out.print("printStackTrace(): ");

e.printStackTrace();

}

}

}

**Output:**

csharp

CopyEdit

getMessage(): / by zero

toString(): java.lang.ArithmeticException: / by zero

printStackTrace(): java.lang.ArithmeticException: / by zero

at ExceptionMethodsDemo.main(ExceptionMethodsDemo.java:4)

**Conclusion:**

These methods help developers understand, debug, and manage exceptions effectively during program execution.

**2. Explain the different scenarios causing ―Exception in thread main‖**

**Introduction:**

The message **"Exception in thread 'main'"** indicates that an **uncaught exception occurred in the main() method**, which is the entry point of any Java application.

**Common Scenarios:**

**1. Division by Zero**

java

CopyEdit

public class Demo {

public static void main(String[] args) {

int x = 5 / 0;

}

}

**Output:**  
Exception in thread "main" java.lang.ArithmeticException: / by zero

**2. ArrayIndexOutOfBoundsException**

java

CopyEdit

public class Demo {

public static void main(String[] args) {

int[] arr = new int[3];

System.out.println(arr[5]);

}

}

**Output:**  
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException

**3. NullPointerException**

java

CopyEdit

public class Demo {

public static void main(String[] args) {

String s = null;

System.out.println(s.length());

}

}

**Output:**  
Exception in thread "main" java.lang.NullPointerException

**4. ClassNotFoundException (in class loading)**

java

CopyEdit

public class Demo {

public static void main(String[] args) throws Exception {

Class.forName("com.abc.NonExistentClass");

}

}

**5. NumberFormatException**

java

CopyEdit

public class Demo {

public static void main(String[] args) {

int num = Integer.parseInt("abc");

}

}

**Conclusion:**

The "Exception in thread main" error is **not a separate exception type**—it is a message printed by JVM indicating that the main method has thrown an uncaught exception.

**3. How will you create your Own Exception Subclasses?**

**Introduction:**

Java allows users to define **custom exception classes** by **extending the Exception class**. These are called **user-defined exceptions**.

**Steps to Create Custom Exception:**

1. Extend the Exception class.
2. Provide constructors (default and parameterized).
3. Optionally override methods like toString() or getMessage().

**Example Program:**

java

CopyEdit

// Step 1: Define custom exception

class AgeInvalidException extends Exception {

AgeInvalidException(String msg) {

super(msg);

}

}

// Step 2: Use it in application

public class CustomExceptionDemo {

static void checkAge(int age) throws AgeInvalidException {

if (age < 18)

throw new AgeInvalidException("Age must be 18 or above.");

else

System.out.println("Eligible to vote.");

}

public static void main(String[] args) {

try {

checkAge(15);

} catch (AgeInvalidException e) {

System.out.println("Caught Exception: " + e.getMessage());

}

}

}

**Output:**

php

CopyEdit

Caught Exception: Age must be 18 or above.

**Advantages of Custom Exceptions:**

* Provide meaningful error messages.
* Help distinguish between different error types.
* Improve code readability and debugging.

**Conclusion:**

Custom exceptions allow developers to handle application-specific errors gracefully, making the application more robust and maintainable.

**UNIT IV I/O, GENERICS, STRING HANDLING**

**1. Describe the creation of a single thread and multiple threads using an example. (Apr/May 2019)**

**🧠 Introduction:**

A **thread** is a lightweight sub-process, the smallest unit of processing. Multithreading in Java allows concurrent execution of two or more threads.

**✅ Two ways to create threads:**

1. **By extending the Thread class**
2. **By implementing the Runnable interface**

**🔹 Single Thread Example using Thread class:**

java

CopyEdit

class MyThread extends Thread {

public void run() {

System.out.println("Single thread is running...");

}

public static void main(String[] args) {

MyThread t1 = new MyThread(); // Create object

t1.start(); // Start the thread

}

}

**🔹 Multiple Threads Example using Runnable Interface:**

java

CopyEdit

class MyRunnable implements Runnable {

public void run() {

System.out.println("Thread running: " + Thread.currentThread().getName());

}

public static void main(String[] args) {

Thread t1 = new Thread(new MyRunnable(), "Thread-1");

Thread t2 = new Thread(new MyRunnable(), "Thread-2");

t1.start(); // Start first thread

t2.start(); // Start second thread

}

}

**📌 Output (Sample):**

mathematica

CopyEdit

Thread running: Thread-1

Thread running: Thread-2

**💡 Conclusion:**

Java provides powerful thread handling features that allow us to create and manage multiple threads for concurrent execution.

**2. With illustrations explain multithreading, interrupting threads, thread states and thread properties.**

**🔹 Multithreading:**

Multithreading allows a program to execute two or more parts of a program concurrently. Each part is called a thread.

**🔹 Key Concepts in Multithreading:**

| **Concept** | **Description** |
| --- | --- |
| **Thread** | Smallest unit of execution |
| **Main Thread** | The initial thread created when the program starts |
| **Daemon Thread** | Background thread that ends when all user threads end |
| **Priority** | Each thread has priority (1 to 10), default is 5 |
| **Name** | Can be set with setName() and retrieved with getName() |

**Example: Setting Name and Priority**

java

CopyEdit

class MyThread extends Thread {

public void run() {

System.out.println("Thread name: " + getName());

System.out.println("Thread priority: " + getPriority());

}

public static void main(String[] args) {

MyThread t1 = new MyThread();

t1.setName("WorkerThread");

t1.setPriority(Thread.MAX\_PRIORITY);

t1.start();

}

}

**🔹 Interrupting Threads:**

* **interrupt()** method is used to interrupt a thread.
* It sets the **interrupted flag** to true.

java

CopyEdit

class MyThread extends Thread {

public void run() {

try {

for (int i = 1; i <= 5; i++) {

System.out.println("Running " + i);

Thread.sleep(1000);

}

} catch (InterruptedException e) {

System.out.println("Thread interrupted!");

}

}

public static void main(String[] args) {

MyThread t1 = new MyThread();

t1.start();

t1.interrupt(); // Interrupt the thread

}

}

**🔹 Thread States:**

| **State** | **Description** |
| --- | --- |
| **New** | Thread is created but not started |
| **Runnable** | Thread is ready to run |
| **Running** | Thread is executing |
| **Blocked/Waiting** | Thread is waiting for a resource |
| **Terminated** | Thread has completed execution |

**Thread State Illustration:**

plaintext

CopyEdit

New → start() → Runnable → Running → (sleep/wait) → Blocked → Running → Terminated

**✅ Conclusion:**

Thread properties and state management help build efficient multithreaded applications that are responsive and performant.

**3. Describe the life cycle of thread and various thread methods. Or Explain in detail the different states of a thread?**

**🔁 Thread Life Cycle Diagram:**

sql

CopyEdit

New

|

start()

|

Runnable

|

Thread Scheduler

↓

Running

/ \

sleep() wait()

↓ ↓

Timed Waiting / Blocked

↓

Runnable again

↓

Terminated (Dead)

**🔹 Thread States:**

| **State** | **Description** |
| --- | --- |
| **New** | Thread object is created but not started. |
| **Runnable** | After calling start(), thread is ready to run. |
| **Running** | Thread is currently executing. |
| **Blocked** | Waiting for a monitor lock. |
| **Waiting** | Waiting indefinitely for another thread to perform a specific action. |
| **Timed Waiting** | Waits for a specified time. |
| **Terminated** | The thread has finished execution. |

**🔧 Important Thread Methods:**

| **Method** | **Description** |
| --- | --- |
| start() | Starts a thread. |
| run() | Entry point of a thread. |
| sleep(ms) | Pauses thread for a specific time. |
| join() | Waits for a thread to finish. |
| interrupt() | Interrupts the thread. |
| isAlive() | Checks if the thread is active. |
| setName() / getName() | Sets/gets the thread name. |
| setPriority() / getPriority() | Sets/gets thread priority. |

**✅ Example: Thread Life Cycle in Action**

java

CopyEdit

class MyThread extends Thread {

public void run() {

System.out.println("Thread is running...");

}

public static void main(String[] args) throws InterruptedException {

MyThread t = new MyThread(); // NEW

System.out.println("State after creation: " + t.getState());

t.start(); // RUNNABLE

System.out.println("State after start(): " + t.getState());

t.join(); // TERMINATED

System.out.println("State after termination: " + t.getState());

}

}

**UNIT V JAVAFX EVENT HANDLING, CONTROLS AND COMPONENTS**

**1. What is Event Delegation Model and what are the event classes and event interfaces?**

**🔹 Event Delegation Model (EDM):**

The **Event Delegation Model** in Java is a mechanism that handles events such as button clicks, mouse movement, keyboard input, etc. It is based on the concept of event **source**, **listener**, and **event object**.

**✅ How It Works:**

* **Source**: The component (like a button) that generates an event.
* **Event Object**: Encapsulates information about the event.
* **Listener**: An object that listens for and handles events.

The source generates an event and sends it to the listener, which then processes the event.

**🔁 Diagram of Event Delegation Model:**

lua

CopyEdit

+-----------+ generates +---------------+ handled by +-------------+

| Source | -----------------------> | Event Object | ---------------------> | Listener |

| (Button) | | (ActionEvent) | | (ActionListener)

+-----------+ +---------------+ +-------------+

**🧱 Important Event Classes in AWT:**

| **Event Class** | **Description** |
| --- | --- |
| ActionEvent | Generated by buttons, menu items, etc. |
| ItemEvent | Generated by checkboxes, choices, etc. |
| MouseEvent | Mouse actions like click, press, release |
| KeyEvent | Keyboard input |
| WindowEvent | Window open, close, minimize, etc. |
| FocusEvent | Component gains or loses focus |
| TextEvent | Generated when the text changes |

**🔧 Event Interfaces in Java:**

| **Interface** | **Methods** |
| --- | --- |
| ActionListener | actionPerformed(ActionEvent e) |
| ItemListener | itemStateChanged(ItemEvent e) |
| MouseListener | mouseClicked(), mousePressed()... |
| KeyListener | keyPressed(), keyReleased()... |
| WindowListener | windowClosing(), windowOpened()... |

**🔤 Example: ActionListener with Button**

java

CopyEdit

import java.awt.\*;

import java.awt.event.\*;

class MyFrame extends Frame implements ActionListener {

Button b;

MyFrame() {

b = new Button("Click Me");

b.setBounds(100, 100, 80, 30);

b.addActionListener(this); // register listener

add(b);

setSize(300, 300);

setLayout(null);

setVisible(true);

}

public void actionPerformed(ActionEvent e) {

b.setLabel("Clicked!");

}

public static void main(String[] args) {

new MyFrame();

}

}

**🧠 Conclusion:**

The **Event Delegation Model** is efficient and clean. It separates event source and handling, promoting better modularity in GUI development.

**2. Explain various components in AWT**

**🧱 AWT Components Overview:**

Java AWT (Abstract Window Toolkit) provides many **UI components** to design graphical interfaces such as buttons, labels, text fields, checkboxes, etc.

**🔹 Categories of AWT Components:**

| **Category** | **Components** |
| --- | --- |
| **Labeling** | Label – displays a text string |
| **Input Fields** | TextField, TextArea – for inputting single/multi-line text |
| **Buttons** | Button, Checkbox, Radio Button (via CheckboxGroup) |
| **Lists** | List, Choice – for selecting items |
| **Containers** | Panel, Frame, Dialog, Window |
| **Layout Managers** | FlowLayout, BorderLayout, GridLayout, CardLayout |
| **Menus** | Menu, MenuBar, MenuItem |
| **Events** | Action, Mouse, Key, Focus, Window |

**✅ Common AWT Components:**

| **Component** | **Description** |
| --- | --- |
| Label | Displays a string |
| Button | Triggers actions when clicked |
| TextField | Single-line input box |
| TextArea | Multi-line input box |
| Checkbox | Toggle input |
| Choice | Drop-down list |
| List | Scrollable list |
| Canvas | Custom drawing |
| Scrollbar | Adds horizontal/vertical scroll |
| Panel | Container to group components |
| Frame | Main window of an AWT application |

**🎨 Example: Using Basic AWT Components**

java

CopyEdit

import java.awt.\*;

public class AWTExample {

public static void main(String[] args) {

Frame f = new Frame("AWT Example");

Label label = new Label("Enter your name:");

label.setBounds(50, 50, 120, 30);

TextField tf = new TextField();

tf.setBounds(180, 50, 100, 30);

Button b = new Button("Submit");

b.setBounds(100, 100, 80, 30);

f.add(label); f.add(tf); f.add(b);

f.setSize(400, 200);

f.setLayout(null);

f.setVisible(true);

}

}

**🧠 Conclusion:**

AWT provides a wide variety of UI components, allowing Java developers to build interactive and responsive GUI applications.

**3. State and explain the basics of AWT Event handling in detail (Nov/Dec 2018)**

**🧠 Introduction:**

AWT uses the **Event Delegation Model** for handling events. Each component (like a button) generates events. Listeners handle those events.

**✅ AWT Event Handling Steps:**

1. **Import java.awt.event**
2. **Implement listener interface**
3. **Override event-handling method**
4. **Register listener with component using addXXXListener()**

**🔄 Event Flow Diagram:**

pgsql

CopyEdit

User Action → Event Generated → Event Object Created → Sent to Listener → Handler Method Invoked

**🧱 Common Listener Interfaces:**

| **Listener Interface** | **Method** |
| --- | --- |
| ActionListener | actionPerformed() |
| ItemListener | itemStateChanged() |
| MouseListener | mouseClicked(), mousePressed()... |
| KeyListener | keyPressed(), keyTyped() |
| WindowListener | windowClosing(), etc. |

**🎯 Example: Handling Button Click Using ActionListener**

java

CopyEdit

import java.awt.\*;

import java.awt.event.\*;

public class EventDemo extends Frame implements ActionListener {

Button b;

TextField tf;

EventDemo() {

tf = new TextField();

tf.setBounds(60, 50, 170, 20);

b = new Button("Click Me");

b.setBounds(100, 100, 80, 30);

b.addActionListener(this); // Register the event listener

add(b); add(tf);

setSize(300, 200);

setLayout(null);

setVisible(true);

}

public void actionPerformed(ActionEvent e) {

tf.setText("Button Clicked!");

}

public static void main(String[] args) {

new EventDemo();

}

}