JAVA PROGRAMMING

Unit:- 1

@@@@Introduction to Java

Java is one of the most widely used programming languages in the world, created by Sun Microsystems (now owned by Oracle) in 1995. It was designed to be platform-independent, secure, and easy to use. Java quickly became popular for developing web applications, mobile apps (especially Android), and large-scale enterprise systems due to its simplicity and portability.

Java History :1991: Java was initially developed by James Gosling, Mike Sheridan, and Patrick Naughton at Sun Microsystems under the name "Oak," aiming to create a language for consumer electronics.**1995:** It was officially named Java and released as a core technology for internet-based applications. The phrase "Write Once, Run Anywhere" (WORA) became the language's key selling point, emphasizing its portability across different platforms.**2009:** Oracle acquired Sun Microsystems, gaining control of Java.

Java Features

Simple: Java has a clean syntax, and many complex features (like pointers) are removed to simplify development.**Object-Oriented**: Java is based on the object-oriented programming paradigm, emphasizing modular code and reusable objects.**Platform-Independent**: Java code is compiled into bytecode that runs on the Java Virtual Machine (JVM), making it platform-independent.**Distributed Computing**: Java provides a set of APIs that make it easy to write distributed applications (like web apps).**Secure**: Java has built-in security features like bytecode verification, runtime security, and the sandboxing model for applets.**Multithreaded**: Java supports multithreading, enabling efficient execution of multiple tasks simultaneously.**Robust**: Java handles errors, memory management (through garbage collection), and exception handling, making it a reliable language for large-scale applications.

Java and Internet

Java's design is optimized for the internet. Features like the ability to create distributed applications, secure transactions, and platform independence are key for web development.**Java Applets**: These are small programs that run inside a web browser, delivering interactive content.**Java Servlets**: Java servlets run on the server side and generate dynamic content in response to client requests.**Web Services**: Java can be used to create RESTful or SOAP-based web services, enabling different applications to communicate over the internet.

Java and World Wide Web

Java has a long history of working with the World Wide Web, especially with:**Servlets and JSP**: Java's Servlets and JavaServer Pages (JSP) allow the development of dynamic, interactive web applications. Servlets handle requests from browsers, while JSPs generate the HTML content dynamically.**Platform Independence**: Java's bytecode allows web applications to run seamlessly across different operating systems, which is ideal for web-based applications.**Security**: Java has an array of security features, such as the Java sandbox, that ensure safe execution of code, making it trustworthy for online use.

Java Program Structure

A Java program generally follows this structure:**Package Declaration**: It defines the location of the class file in the directory structure.**Import Statements**: Used to import other Java packages that the program uses.**Class Declaration**: Every Java program has at least one class.**Main Method**: The entry point of a Java program, `public static void main(String[] args)`.**Statements and Expressions**: The code that is executed, like variable declarations and operations.

Example:

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

Java Tokens

Java tokens are the smallest units in the Java language. These include:**Keywords**: Reserved words like `class`, `public`, `if`, `for`, etc.**Identifiers**: Names used for variables, functions, classes, etc.**Literals**: Fixed values like integers (`10`), characters (`'a'`), strings (`"Hello"`), etc.**Operators**: Symbols like `+`, `-`, `*`, `/`, `==`, `&&`, etc.**Separators**: Punctuation marks like commas (`,`), semicolons (`;`), parentheses (`()`), etc.

Java Virtual Machine (JVM)

The **JVM** is the engine that executes Java bytecode. Java programs are compiled into bytecode by the Java compiler, which is platform-independent. The JVM reads and executes this bytecode on any system that has a JVM installed, ensuring the "Write Once, Run Anywhere" principle.

Key functions of JVM:**Loading Code**: The JVM loads compiled Java bytecode.**Bytecode Verification**: It checks the bytecode for security violations.**Execution**: It translates the bytecode into machine code specific to the platform.**Garbage Collection**: JVM automatically manages memory by reclaiming unused objects.

Data Types in Java

Java supports two categories of data types:

Primitive Data Types: These are predefined types with specific sizes and ranges. `byte`: 1 byte, `short`: 2 bytes, `int`: 4 bytes , `long`: 8 bytes , `float`: 4 bytes , `double`: 8 bytes, `char`: 2 bytes (for storing single characters), `boolean`: 1 bit (either `true` or `false`)

Reference Data Types: These refer to objects or arrays. They store the memory address of the data.**Classes**: A reference type that can store any object.**Arrays**: A reference type that holds a fixed-size collection of elements.**Interfaces**: Reference type for defining abstract methods that other classes must implement.

Operators and Expressions in Java

Operators are symbols used to perform operations on variables and values. Java has the following types:

1. **Arithmetic Operators**: `+`, `-`, `*`, `/`, `%` for mathematical operations.**Relational Operators**: `==`, `!=`, `>`, `<`, `>=`, `<=` for comparisons.**Logical Operators**: `&&`, `||`, `!` for logical operations.**Assignment Operator**: `=` for assigning values.**Unary Operators**: `++`, `--`, `+`, `-` for single operand operations.**Bitwise Operators**: `&`, `|`, `^`, `~`,

<<, >>, >>> for manipulating individual bits.**Ternary Operator**: `condition ? trueValue : falseValue`.**Expressions** are combinations of variables, operators, and values that Java evaluates to produce a result.

Example:`int a = 10;`

```
int b = 5;
int result = a + b * 2;  // Expression
```

This was an overview of Java programming fundamentals. These concepts serve as the building blocks for developing applications in Java.

@@@@Decision Making and Branching in Java

In Java, decision-making is done using **if-else** statements, **switch** statements, and other control flow mechanisms. These help to control the flow of execution based on conditions.

1. If Statement

The `if` statement checks if a condition is true and executes the code inside the block if it is true.

Example: `int number = 10;`

```
if (number > 0) {
    System.out.println("The number is
positive.");
}
```

2. If-else Statement

If the condition is false, the `else` block is executed.

Example: `int number = -5;`

```
if (number > 0) {
    System.out.println("The number is
positive.");
} else {
    System.out.println("The number is
negative.");
}
```

3. Else-if Ladder

Multiple conditions are checked using `else-if`.

Example: `int number = 0;`

```
if (number > 0) {
    System.out.println("The number is
positive.");
} else if (number < 0) {
    System.out.println("The number is
negative.");
} else {
    System.out.println("The number is
zero.");
}
```

4. Switch Statement

A more efficient way to handle multiple conditions based on a single variable's value.

Example: `int day = 2;`

```
switch (day) {
    case 1: System.out.println("Monday");
break;
    case 2: System.out.println("Tuesday");
break;
    case 3: System.out.println("Wednesday");
break;
    default: System.out.println("Invalid
day");
}
```

Looping in Java

Looping allows you to repeat a block of code multiple times. Java supports several loop structures:

1. For Loop

Used when you know how many times you want to loop.

Example: `for (int i = 0; i < 5; i++) {`

```
    System.out.println(i);  // Prints 0 to 4
}
```

2. While Loop

Used when you want to loop while a condition is true.

Example: `int i = 0;`

```
while (i < 5) {
    System.out.println(i);  // Prints 0 to 4
    i++;
}
```

3. Do-While Loop

Similar to the `while` loop, but the code inside the loop is executed at least once.

Example: `int i = 0;`

```
do {
    System.out.println(i);  // Prints 0 to 4
    i++;
} while (i < 5);
```

Classes and Methods in Java

In Java, **classes** are blueprints for creating objects, and **methods** are the actions that objects can perform.

1. Defining a Class

A class is defined using the `class` keyword.

Example: `class Car {`

```
    String model;
    int year;

    void start() {
        System.out.println("The car is
starting.");
    }

    void stop() {
        System.out.println("The car is
stopping.");
    }
}
```

2. Defining a Method

Methods are functions that belong to classes. They define behavior for the objects of the class.

Example: `class Calculator {`

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

3. Creating Objects

Objects are instances of classes, created using the `new` keyword.

Example: `Car car = new Car();  // Create an object of the Car class`

```
car.start();           // Call the start
method
```

@@@@@Inheritance in Java

Inheritance is a mechanism where one class inherits the properties (fields) and behaviors (methods) from another class, enabling code reuse and establishing a relationship between the parent and child classes.

1. Using Existing Classes

Inheritance allows you to use an existing class and extend its functionality by creating a new class. Example: `class Animal {`

```
    void eat() {
        System.out.println("This animal eats
food.");
    }
}

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

2. Class Inheritance

The `extends` keyword is used to inherit from a parent class. A subclass can access the public and protected members of its superclass. Example: `class Vehicle {`

```
    void start() {
        System.out.println("The vehicle is
starting.");
    }
}

class Car extends Vehicle {
    void stop() {
        System.out.println("The car is
stopping.");
    }
}
```

3. Choosing Base Class

When designing classes, choose a base class that provides common behavior and attributes that can be inherited. For example, a `Shape` class might be the base class for specific shapes like `Circle`, `Square`, etc.

4. Accessing Attributes of Superclass

In a subclass, you can access attributes from the superclass. If the superclass members are `public` or `protected`, they can be accessed directly; if they're `private`, they need getters and setters. Example: `class Animal {`

```
    private String name;

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}

class Dog extends Animal {
    void display() {
        System.out.println("Dog's name is: "
+ getName());  // Accessing superclass method
    }
}
```

5. Types of Inheritance

Java supports the following types of inheritance:

Single Inheritance: A subclass inherits from one superclass.Example: `class Dog extends Animal`**Multilevel Inheritance**: A class can inherit from another class, which itself is derived from a parent class.Example: `class Dog extends Animal extends Mammal`**Hierarchical Inheritance**: Multiple subclasses inherit from a single superclass.Example: `class Dog extends Animal, class Cat extends Animal`**Multiple Inheritance (interface-based)**: In Java, multiple inheritance is achieved through interfaces (as Java doesn't support it directly with classes).
Example: `class Dog implements Animal, Pet`

6. Abstract Classes

An abstract class is a class that cannot be instantiated and is meant to be subclassed. It can have abstract methods (methods without body) that must be implemented by subclasses. Example: `abstract class Animal {`

```
    abstract void sound();  // Abstract
method
```

```
    void eat() {
        System.out.println("This animal eats
food.");
    }
}

class Dog extends Animal {
    void sound() {
        System.out.println("The dog barks.");
    }
}
```

7. Using Final Modifier

The `final` modifier can be used with classes, methods, and variables.**Final Class**: A class marked as `final` cannot be subclassed. `final class Animal { }`

- **Final Method**: A method marked as `final` cannot be overridden in a subclass.

```
class Animal {
     final void eat() {
          System.out.println("This animal
   eats food.");
    }
   }
```

Final Variable: A variable marked as `final` cannot be reassigned after initialization. `final int x = 10;`

Unit:- 2

@@@@ Polymorphism in Java

Polymorphism is one of the core concepts of Object-Oriented Programming (OOP), allowing objects of different classes to be treated as objects of a common superclass. The most important benefit of polymorphism is that it allows you to write more flexible and reusable code.

Types of Polymorphism

Compile-time Polymorphism (Static Polymorphism):
This type of polymorphism is resolved at **compile-time** and is achieved using method overloading or operator overloading. # **Method Overloading**: This occurs when multiple methods in the same class have the same name but different parameter types or numbers. The appropriate method is chosen at compile time based on the method signature.

Example: `class Calculator {`

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

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

public class Main {
    public static void
main(String[] args) {
        Calculator calc = new
Calculator();
        System.out.println(calc.add(10,
20));  // Calls the int version

        System.out.println(calc.add(10.
5, 20.5));  // Calls the double
version
    }
}
```

In this case, the `add` method is overloaded based on the parameter type, which is resolved at compile time.

Runtime Polymorphism (Dynamic Polymorphism):
This type of polymorphism is resolved at **runtime** and is achieved through method overriding. It occurs when a subclass provides a specific implementation of a method that is already defined in the superclass.

Method Overriding: A method in a subclass overrides a method in its superclass with the same method signature. The version of the method that gets called depends on the object's runtime type, not the reference type. Example: `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
Dog(); // Animal reference, Dog
object
        myAnimal.sound();  //
Output: Dog barks

        myAnimal = new Cat();
// Animal reference, Cat object
        myAnimal.sound();  //
Output: Cat meows
    }
}
```

In the example above, the method `sound` is overridden in both `Dog` and `Cat` classes. The actual method that gets called depends on the object's runtime type (`Dog` or `Cat`), even though the reference is of type `Animal`. This is an example of **runtime polymorphism**.

@@@@ Packages in Java

A **package** in Java is a way to group related classes and interfaces together. It helps organize code into namespaces, making it easier to manage large codebases.

Understanding Packages

Organize Code: Packages allow you to group related classes, interfaces, and sub-packages to make the code modular and manageable.**Avoid Naming Conflicts**: They help avoid naming conflicts by differentiating between classes with the same name but in different packages.**Access Control**: They can also control access to classes, methods, and variables.

Types of Packages

Built-in Packages: Java comes with several built-in packages, like `java.util`, `java.io`, and `java.lang`, which provide commonly used functionality such as collections, file handling, and system operations.**User-defined Packages**: Developers can create their own packages to logically group classes and interfaces.

Defining a Package

To define a package, the `package` keyword is used at the very beginning of a Java source file. All classes that belong to the same package should have this declaration.Example: `package com.example.myapp;`

```
class MyClass {
    void display() {
        System.out.println("Hello from
MyClass!");
    }
}
```

In this example, `MyClass` is part of the `com.example.myapp` package. The package name must match the directory structure where the class file is located.

Packaging up Your Classes

Once you define a package, you can place your classes in that package directory. When compiling and running Java programs that use packages, you need to follow the structure:**Directory Structure**: The directory structure should reflect the package name.For the package `com.example.myapp`, the file should be in a directory path like `com/example/myapp/`.**Compilation**: When compiling classes in a package, make sure you're in the parent directory of the root package and use the `javac` command. For example: `javac com/example/myapp/MyClass.java`

Adding Classes from a Package to Your Program

To use classes from a package, you need to **import** them into your program. The `import` keyword is used for this.

Importing Classes

Single Class Import:
To import a specific class from a package: `import com.example.myapp.MyClass;`

```
   public class Main {
public static void main(String[] args) {
MyClass myObject = new MyClass();
   myObject.display();
}
```

```
}
```

Importing All Classes from a Package:
You can import all the classes in a package using the `*` wildcard.

```
import com.example.myapp.*;

public class Main {
    public static void main(String[]
args) {
        MyClass myObject = new MyClass();
        myObject.display();
    }
}
```

@@@@ Interfaces in Java

An **interface** in Java is a contract that specifies a set of methods that a class must implement. Interfaces allow multiple inheritance of method signatures, which is particularly useful in Java since the language doesn't support multiple inheritance with classes.

Defining an Interface

An interface is defined using the `interface` keyword. It can only have abstract methods (methods without a body), constants, default methods, and static methods.

Example: `interface Animal {`

```
    void sound();  // Abstract method

    default void sleep() {
        System.out.println("The animal is
sleeping.");
    }
}

class Dog implements Animal {
    @Override
    public void sound() {
        System.out.println("The dog barks.");
    }
}

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

Output:

```
The dog barks.
The animal is sleeping.
```

In the example above:The `Animal` interface defines an abstract method `sound()` and a default method `sleep()`.The `Dog` class implements the `Animal` interface, providing its own implementation of the `sound()` method and inheriting the `sleep()` method.

@@@@ Understanding CLASSPATH in Java

In Java, the **CLASSPATH** is an environment variable that tells the Java Virtual Machine (JVM) and Java compiler where to find the classes and packages that are used in a Java program. It plays a key role in locating class files when running or compiling Java applications.

How CLASSPATH Works

The **CLASSPATH** specifies the directories, JAR files, and ZIP files that contain the classes that the JVM needs to locate when running a program.It can be set at the command line or through the system environment variables.

Setting the CLASSPATH

For a Single Class File: If your class files are located in a specific directory, you can set the CLASSPATH for that directory. Example: `export CLASSPATH=/home/user/classes`……This will make the JVM look for class files in the `/home/user/classes` directory.

Using JAR Files: If you are using external libraries in the form of JAR (Java ARchive) files, you can add them to the CLASSPATH. Example: `export CLASSPATH=/home/user/libs/myLibrary.jar`

Multiple Entries: You can add multiple directories or JAR files to the CLASSPATH, separating them by a colon (`:`) on Linux/macOS or a semicolon (`;`) on Windows. Example (Linux/macOS):

```
export CLASSPATH=/home/user/classes:/home/user/libs/myLibrary.jar
```

Example (Windows): `set CLASSPATH=C:\Users\user\classes;C:\Users\user\libs\myLibrary.jar`

1. **Using the CLASSPATH in Java Commands:**
   You can set the CLASSPATH directly when compiling or running a Java program using the `-classpath` or `-cp` option.

   Example: `javac -cp .:/home/user/libs/myLibrary.jar MyProgram.java`

   ```
   java -cp .:/home/user/libs/myLibrary.jar MyProgram
   ```

   Here, `.` represents the current directory, and the `-cp` flag adds the current directory and the external JAR file to the classpath.

@@@# Access Protection in Packages

In Java, access protection mechanisms allow you to control the visibility of classes, methods, and fields from other classes. This helps you implement **encapsulation** and hide the implementation details.

Access Modifiers

There are four main types of access modifiers in Java:

Public Access Modifier (`public`): The class, method, or field is accessible from anywhere in the program.If a class is declared `public`, it can be accessed from any other class.Example:

```
public class MyClass {
    public int number;
}
```

Private Access Modifier (`private`): The class member is accessible only within the class in which it is declared. It cannot be accessed from outside the class, even by subclasses. Example:

```
class MyClass {
                  private int number;
                  private void display() {
        System.out.println(number);
                  }
             }
```

Protected Access Modifier (`protected`) : The class member is accessible within the class, within subclasses, and within the same package. Example: `class Animal {`

```
    protected void makeSound() {
        System.out.println("Animal sound");
    }
}

class Dog extends Animal {
    public void display() {
        makeSound();  // Accessing protected
method from superclass
    }
}
```

Default Access Modifier (Package-Private) : If no access modifier is specified, the member is accessible only within the same package. This is also known as **package-private** access. Example: `class MyClass {`

```
              void display() {  // No
           modifier means package-private

           System.out.println("Hello");
               }
           }
```

Access Control in Packages

If a class or member is declared `private`, it is not accessible from any other class, even if they are in the same package.If a class or member is `protected`, it is visible to other classes in the same package or subclasses in other packages.If no modifier is used (i.e., package-private), it is only accessible within the same package.If a class or method is `public`, it can be accessed from any package.

Concept of Interface in Java

An **interface** in Java defines a contract of methods that a class must implement. It specifies what a class should do, but not how it should do it. Interfaces are used to achieve **abstraction** and allow multiple inheritance in Java (though Java doesn't support multiple inheritance with classes directly).

Defining an Interface

An interface is defined using the `interface` keyword. It can contain:**Abstract methods** (methods without a body).**Constants** (by default, all fields in an interface are `public`, `static`, and

final).**Default methods** (methods with a body, introduced in Java 8).**Static methods** (methods with a body that belong to the interface itself). **Private methods** (for internal use within the interface, introduced in Java 9).

Example of an interface: interface Animal {

```
    // Abstract method
    void sound();  // No body, just the
method signature

    // Default method (Java 8+)
    default void sleep() {
        System.out.println("The animal is
sleeping.");
    }

    // Static method (Java 8+)
    static void breath() {
        System.out.println("The animal is
breathing.");
    }
}
```

Implementing an Interface

A class that implements an interface must provide implementations for all the abstract methods declared in the interface. Example of a class implementing an interface: class Dog implements Animal {

```
  // Implementing the abstract method from
Animal interface
    public void sound() {
        System.out.println("The dog barks.");
    }
}
```

Using an Interface

Once an interface is implemented by a class, the methods can be called on objects of that class. Example:

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

        // Calling static method of interface
        Animal.breath();  // Output: The
animal is breathing.
    }
}
```

Multiple Interfaces and Multiple Inheritance

A class can implement multiple interfaces, which is Java's way of allowing multiple inheritance of method signatures. However, it cannot inherit multiple classes (since Java does not support multiple inheritance with classes). Example of a class implementing multiple interfaces: interface Animal {

```
    void sound();
}

interface Mammal {
    void move();
}

class Dog implements Animal, Mammal {
    public void sound() {
        System.out.println("The dog barks.");
    }

    public void move() {
        System.out.println("The dog runs.");
    }
}
```

@@@@ Exception Handling in Java

Exception handling in Java is a mechanism to handle runtime errors (or exceptions) so that the normal flow of the application can be maintained. Java provides a robust mechanism for handling exceptions using the try, catch, throw, throws, and finally keywords.

Types of Exceptions in Java

There are two main types of exceptions in Java:

Checked Exceptions: These are exceptions that are checked at **compile time**. These exceptions are subclasses of Exception but not of RuntimeException. The compiler forces you to either handle these exceptions using a try-catch block or declare them using the throws keyword. Example: IOException, SQLException.

Unchecked Exceptions: These are exceptions that are not checked at **compile time**. They are usually runtime exceptions and are subclasses of RuntimeException. These exceptions usually occur due to programming bugs, such as division by zero, null pointer dereference, etc. Example: NullPointerException, ArithmeticException, ArrayIndexOutOfBoundsException. Unchecked exceptions can be optionally handled; they are not required to be caught or declared.

Errors: Errors are serious issues that generally cannot be handled by the application. These are not exceptions. Errors are subclasses of Error, and they represent severe problems that are usually outside the control of the program. Example: OutOfMemoryError, StackOverflowError.

Dealing with Exceptions in Java

Java provides several mechanisms to handle exceptions effectively. The key components are:

1. Try-Catch Block

The try block contains the code that might throw an exception. The catch block handles the exception if it occurs. try {

```
    int result = 10 / 0;  // This will throw
an ArithmeticException
} catch (ArithmeticException e) {
    System.out.println("Error: " +
e.getMessage());
}
```

In the example above, the code inside the try block throws an ArithmeticException (due to division by zero), which is then caught in the catch block, and an appropriate message is printed.

2. Throwing an Exception (Using `throw`)

The `throw` keyword is used to explicitly throw an exception. You can throw both checked and unchecked exceptions.

Example:
```
public class Example {

    public static void checkAge(int age) {
        if (age < 18) {
            throw new
IllegalArgumentException("Age must be at
least 18.");
        } else {
            System.out.println("Access
granted.");
        }
    }

    public static void main(String[] args) {
        checkAge(15);  // This will throw an
exception
    }
}
```

Here, an `IllegalArgumentException` is thrown manually if the age is less than 18.

3. Throws Keyword

The `throws` keyword is used in the method signature to declare that a method may throw one or more exceptions. This allows the caller of the method to handle the exception.

Example:
```
public class Example {

    public static void readFile() throws
IOException {
        // Some code that might throw
IOException
        throw new IOException("File not
found");
    }

    public static void main(String[] args) {
        try {
            readFile();
        } catch (IOException e) {
            System.out.println("Error: " +
e.getMessage());
        }
    }
}
```

In this example, the `readFile` method declares that it may throw an `IOException`, and the calling code must handle it with a `try-catch` block.

4. Finally Block

The `finally` block is used to execute code that must run regardless of whether an exception is thrown or not. It is commonly used for cleanup activities, such as closing files or database connections.

```
public class Example {
    public static void main(String[] args) {
        try {
            System.out.println("Trying to
divide");
            int result = 10 / 0;
        } catch (ArithmeticException e) {
            System.out.println("Error:
Division by zero.");
        } finally {
            System.out.println("This will
always execute.");
        }
    }
}
```

Output:

```
Trying to divide
Error: Division by zero.
This will always execute.
```

Even if an exception is thrown, the code inside the `finally` block will always run.

Exception Objects in Java

Exception objects are instances of classes that are derived from the `Throwable` class. These objects hold information about the exception that occurred.

Exception Hierarchy

Throwable is the superclass of all errors and exceptions. **Error**: Represents serious problems that the application cannot handle (e.g., `OutOfMemoryError`). **Exception**: Represents exceptions that can be handled (e.g., `IOException`, `ArithmeticException`). **RuntimeException**: A subclass of `Exception` representing unchecked exceptions (e.g., `NullPointerException`).

Creating Exception Objects

You can create exception objects either implicitly by the Java runtime (when an exception is thrown) or explicitly by using the `new` keyword.

Example:
```
Implicit exception creation (when an
error occurs)

try {
    int result = 10 / 0;  // Throws
ArithmeticException implicitly
} catch (ArithmeticException e) {
    System.out.println(e);  // Printing the
exception object
}

// Explicit exception creation
try {
    throw new
IllegalArgumentException("Invalid argument
provided");
} catch (IllegalArgumentException e) {
    System.out.println("Caught: " +
e.getMessage());  // Custom error message
}
```

Common Methods of Exception Objects

1. **`getMessage()`**: Returns a detailed message about the exception.
2. **`getCause()`**: Returns the cause of the exception (if any).

3. **`printStackTrace()`**: Prints the stack trace of the exception to the console, which helps in debugging.
4. **`toString()`**: Returns a string representation of the exception.

Example:

```
try {
    int result = 10 / 0;  //
ArithmeticException
} catch (ArithmeticException e) {
    System.out.println("Exception message: "
+ e.getMessage());
    System.out.println("Exception details: "
+ e.toString());
    e.printStackTrace();  // Print stack
trace
}
```

Output:

```
Exception message: / by zero
Exception details:
java.lang.ArithmeticException: / by zero
java.lang.ArithmeticException: / by zero
    at Example.main(Example.java:4)
```

In this example, the `getMessage()` method provides a message about the error, `toString()` gives the class name and message, and `printStackTrace()` prints the exception's stack trace, which helps locate where the error occurred.

Conclusion

Exception Handling in Java is a powerful mechanism to manage runtime errors and ensure that the program can recover from unexpected situations or terminate gracefully.There are two types of exceptions: **checked** and **unchecked**, and Java provides various constructs like `try`, `catch`, `throw`, `throws`, and `finally` to handle exceptions.**Exception objects** carry detailed information about the exception, such as error messages, causes, and stack traces, which helps in identifying and troubleshooting errors.

By effectively handling exceptions, Java programs become more robust, preventing unexpected crashes and improving the overall user experience.

Unit:- 3

✓@@@@ What is Multithreading in Java?

Multithreading is a **concurrent execution** technique in Java where multiple threads run independently but share the same resources, such as memory. Threads are the smallest unit of a process, and Java allows programs to perform multiple operations simultaneously by utilizing multiple threads. This improves the performance of applications, especially for tasks like **file processing**, **network operations**, or **handling user input/output**.

✓Key Concepts of Multithreading:

Thread: A thread is a lightweight process. It has its own execution path and shares resources with other threads.**Process:** A process is a collection of threads.**Main thread:** Every Java program starts with a single thread, called the **main thread**.Multithreading can be used in various applications where **simultaneous tasks** need to be executed in parallel.

✓Creating and Running Multiple Threads

There are **two ways** to create threads in Java:**By implementing the `Runnable` interface.By extending the `Thread` class.**

1. Implementing the Runnable Interface

You implement the `Runnable` interface and override the `run()` method to define the code to be executed in the thread. `class Task1 implements Runnable {`

```
    public void run() {
        System.out.println("Task 1 is running
in " + Thread.currentThread().getName());
    }
}

class Task2 implements Runnable {
    public void run() {
        System.out.println("Task 2 is running
in " + Thread.currentThread().getName());
    }
}

public class MultiThreadExample {
    public static void main(String[] args) {
        // Creating two thread tasks
        Thread thread1 = new Thread(new
Task1());
        Thread thread2 = new Thread(new
Task2());

        // Starting threads
        thread1.start();
        thread2.start();
    }
}
```

Output:

```
Task 1 is running in Thread-0
Task 2 is running in Thread-1
```

2. Extending the Thread Class

You can create a thread by **extending the `Thread` class** and overriding the `run()` method.

```
class Task1 extends Thread {
    public void run() {
        System.out.println("Task 1 is running
in " + Thread.currentThread().getName());
    }
}

class Task2 extends Thread {
    public void run() {
        System.out.println("Task 2 is running
in " + Thread.currentThread().getName());
    }
}

public class MultiThreadExample {
    public static void main(String[] args) {
        // Creating and starting threads
        Task1 thread1 = new Task1();
        Task2 thread2 = new Task2();

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

Output:

```
Task 1 is running in Thread-0
Task 2 is running in Thread-1
```

✅Thread Communication

In multithreading, it's important for threads to **communicate with each other**. Java provides mechanisms for **inter-thread communication**, which allows threads to coordinate their actions. This can be done using **wait()**, **notify()**, and **notifyAll()** methods, which are used to synchronize threads.

1. `wait()`, `notify()`, and `notifyAll()` Methods

`wait()`: Causes the current thread to release the lock and go into the waiting state until it is notified by another thread.`notify()`: Wakes up a single thread that is waiting on the object's monitor (lock).`notifyAll()`: Wakes up all threads that are waiting on the object's monitor.

Example: Thread Communication

```
class Printer {
    public synchronized void printNumbers() {
        for (int i = 1; i <= 5; i++) {
            System.out.println(i);
            try {
                // Let the other thread print
                Thread.sleep(1000);
            } catch (InterruptedException e)
{
                System.out.println(e);
            }
        }
    }
}

class PrintNumbersThread extends Thread {
    Printer printer;

    PrintNumbersThread(Printer printer) {
        this.printer = printer;
    }

    public void run() {
        printer.printNumbers();
    }
}

public class ThreadCommunicationExample {
    public static void main(String[] args) {
        Printer printer = new Printer();

        // Create two threads
        PrintNumbersThread thread1 = new
PrintNumbersThread(printer);
        PrintNumbersThread thread2 = new
PrintNumbersThread(printer);

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

Explanation:

`Printer` class has a synchronized method `printNumbers()`, ensuring only one thread prints numbers at a time.The threads `thread1` and `thread2` are both trying to execute the `printNumbers()` method, but due to synchronization, they will take turns.

✅Thread Lifecycle

The **life cycle** of a thread in Java includes several states:**New:** The thread is created but not yet started.**Runnable:** The thread is ready to run and waiting for CPU time.**Blocked:** The thread is waiting to acquire a lock or resource.**Waiting:** The thread is waiting for another thread's action (e.g., `wait()` or `join()`).**Terminated:** The thread has finished execution.

✅Thread Synchronization

When multiple threads access shared resources (variables, files, etc.), there's a risk of **data inconsistency**. To prevent this, we use **synchronization** in Java.

Synchronization Example:

```
class Counter {
    private int count = 0;

    public synchronized void increment() {
        count++;
    }

    public synchronized void decrement() {
        count--;
    }

    public int getCount() {
        return count;
    }
}

public class ThreadSynchronizationExample {
    public static void main(String[] args) {
        Counter counter = new Counter();

        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                counter.increment();
            }
        });

        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                counter.decrement();
            }
        });

        t1.start();
        t2.start();

        try {
            t1.join();
            t2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("Final count: " +
counter.getCount());
    }
}
```

@@@@ ✅Input/Output in Java

Input/Output (I/O) in Java refers to the process of **reading data** from external sources (such as files, keyboard, or network) and **writing data** to external destinations (like console, files, or network). Java provides a rich set of I/O APIs to work with both

byte-based and **character-based** streams for efficient data handling.

✓I/O Basics in Java

In Java, the I/O operations are based on **streams**. A stream is a sequence of data that can be either **input** (data coming in) or **output** (data going out).Java provides two types of streams:**Byte Streams**: Handle raw binary data (e.g., image files, audio files).**Character Streams**: Handle data in **character** format (e.g., text files, console input).

✓Byte Stream and Character Stream

1. Byte Streams:

Byte streams are used for **handling I/O of raw binary data** (like image or audio files). These streams read and write **byte-oriented data** (1 byte at a time).Classes like `InputStream` and `OutputStream` are the parent classes for byte streams. **Common Byte Stream Classes:**`FileInputStream`: Reads bytes from a file.`FileOutputStream`: Writes bytes to a file.

Example of **Byte Stream (File I/O)**:

```
import java.io.*;

public class ByteStreamExample {
    public static void main(String[] args) {
        try {
            // Write data to a file using FileOutputStream
            FileOutputStream fos = new FileOutputStream("example.txt");
            fos.write(65);  // Writing byte value 'A' (ASCII code 65)
            fos.close();

            // Read data from a file using FileInputStream
            FileInputStream fis = new FileInputStream("example.txt");
            int data = fis.read();
            System.out.println("Data from file: " + (char) data); // Output: 'A'
            fis.close();
        } catch (IOException e) {
            System.out.println(e);
        }
    }
}
```

2. Character Streams:

Character streams are used for **handling text data**. They work with **Unicode** characters (2 bytes) and are more efficient when working with **text files** compared to byte streams.Classes like `Reader` and `Writer` are the parent classes for character streams. **Common Character Stream Classes:**`FileReader`: Reads characters from a file.`FileWriter`: Writes characters to a file.

Example of **Character Stream (File I/O)**:

```
import java.io.*;

public class CharacterStreamExample {
    public static void main(String[] args) {
        try {
            // Write data to a file using FileWriter
            FileWriter writer = new FileWriter("example.txt");
            writer.write("Hello, World!");
            writer.close();

            // Read data from a file using FileReader
            FileReader reader = new FileReader("example.txt");
            int data;
            while ((data = reader.read()) != -1) {
                System.out.print((char) data);
            }
            reader.close();
        } catch (IOException e) {
            System.out.println(e);
        }
    }
}
```

✓I/O Classes in Java

Java provides a set of classes for handling I/O operations, divided into **byte-based** and **character-based** categories.

Byte Stream Classes:

InputStream: Abstract class for reading byte data. `FileInputStream`, `BufferedInputStream`, `DataInputStream`, `ObjectInputStream`. **OutputStream**: Abstract class for writing byte data. `FileOutputStream`, `BufferedOutputStream` `DataOutputStream`, `ObjectOutputStream`

Character Stream Classes:

Reader: Abstract class for reading character data. `FileReader`, `BufferedReader`, `InputStreamReader`, `CharArrayReader`. **Writer**: Abstract class for writing character data. `FileWriter`, `BufferedWriter`, `OutputStreamWriter`, `CharArrayWriter`

✓Reading and Writing Console Input and Output

Java provides classes like `System.in`, `System.out`, and `Scanner` to read and write data to and from the console.

1. Reading Console Input

Using `Scanner` class: `Scanner` is a useful utility for reading input from various sources, including the console. `import java.util.Scanner;`

```
public class ConsoleInputExample {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter your name: ");
        String name = scanner.nextLine();
        System.out.println("Hello, " + name + "!");
        scanner.close();
    }
}
```

Using `System.in` and `BufferedReader`:
For more low-level input handling, we can use `System.in` and `BufferedReader`.

```
import java.io.*;

public class ConsoleInputUsingBufferedReader
{
    public static void main(String[] args) {
        try {
            BufferedReader reader = new
BufferedReader(new
InputStreamReader(System.in));
            System.out.print("Enter your age:
");
            String age = reader.readLine();
            System.out.println("Your age is:
" + age);
            reader.close();
        } catch (IOException e) {
            System.out.println("Error: " +
e);
        }
    }
}
```

2. Writing Console Output

Using `System.out`: `System.out` is used for output, and the most common method is `System.out.println()`, which prints text to the console.

```
public class ConsoleOutputExample {
    public static void main(String[] args) {
        System.out.println("This is a simple
console output.");
    }
}
```

✓Buffered I/O

Buffered I/O uses buffered streams to improve the performance of I/O operations, particularly when dealing with larger amounts of data. It reads and writes data in chunks (buffers) rather than byte by byte, which reduces the overhead of I/O operations.

BufferedReader (Character Stream) Example:

```
import java.io.*;

public class BufferedReaderExample {
    public static void main(String[] args) {
        try {
            BufferedReader reader = new
BufferedReader(new
FileReader("example.txt"));
            String line;
            while ((line = reader.readLine())
!= null) {
                System.out.println(line);
            }
            reader.close();
        } catch (IOException e) {
            System.out.println(e);
        }
    }
}
```

BufferedOutputStream (Byte Stream) Example:

```
import java.io.*;

public class BufferedOutputStreamExample {
    public static void main(String[] args) {
        try {
            BufferedOutputStream bos = new
BufferedOutputStream(new
FileOutputStream("example.txt"));
            bos.write("Buffered Output Stream
Example".getBytes());
            bos.close();
        } catch (IOException e) {
            System.out.println(e);
        }
    }
}
```

✓Serialization

Serialization is the process of converting an object into a byte stream, which can then be saved to a file or sent over a network. The **`ObjectOutputStream`** class is used to serialize objects, and **`ObjectInputStream`** is used for deserialization. Example of **Serialization**: `import java.io.*;`

```
class Person implements Serializable {
    String name;
    int age;

    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}

public class SerializationExample {
    public static void main(String[] args) {
        try {
            Person person = new
Person("John", 30);
            ObjectOutputStream oos = new
ObjectOutputStream(new
FileOutputStream("person.ser"));
            oos.writeObject(person);
            oos.close();

            ObjectInputStream ois = new
ObjectInputStream(new
FileInputStream("person.ser"));
            Person deserializedPerson =
(Person) ois.readObject();
            ois.close();

            System.out.println("Name: " +
deserializedPerson.name);
            System.out.println("Age: " +
deserializedPerson.age);
        } catch (IOException |
ClassNotFoundException e) {
            System.out.println(e);
        }
    }
}
```

@@@@ ✓ Creating Applets in Java

In Java, an **applet** is a small application that is designed to be embedded in a web page and run in a web browser. Applets are built using Java's **Applet API**. However, it's worth noting that as of recent years, **applets have become obsolete** due to security concerns, and modern web applications have moved to more advanced technologies like HTML5, JavaScript, and CSS. Nevertheless, understanding applets and their life cycle is still useful for historical contexts or legacy applications.

✓Applet Basics

An **applet** is a **Java class** that extends the `Applet` class or implements the `Applet` interface and is used to provide graphical user interfaces (GUIs) within a browser. Applets are typically **embedded** in HTML pages using the `<applet>` tag (though this tag is now deprecated). Key characteristics of an applet: It can be **loaded from the web** and run inside a browser. Applets are designed to be **interactive** and can handle user inputs like **mouse clicks** or **keyboard input**. Applets are run by the **Java Runtime Environment (JRE)** embedded in the browser.

✓Applet Architecture

The architecture of an applet consists of the following components: **Applet Class**: A Java class that extends the `Applet` class or implements the `Applet` interface. This class must include the methods for initializing, starting, and stopping the applet. **HTML Page**: A web page that includes the `<applet>` tag to embed the applet and load it within the browser. The tag specifies the applet class file and various parameters.

Example of an Applet in an HTML page:

```
<applet code="HelloWorldApplet.class"
width="300" height="300">
    Your browser does not support Java
applets.
</applet>
```

Java Plugin: A browser plugin or Java Web Start that allows Java code to run in browsers. Java's **Java Plug-in** has been deprecated, and browsers no longer support applets.

✓Applet Life Cycle

The life cycle of an applet consists of several stages, each controlled by specific methods. These methods are invoked automatically by the applet's environment (usually the web browser or applet viewer). The key methods in the applet life cycle are: **`init()`**: This method is called **once** when the applet is first loaded.It is used for **initialization** tasks, such as setting up resources (like images or data). **Example**: Setting up UI components, initializing variables. `public void init() {`

```
        // Initialize the applet
        System.out.println("Applet
    Initialized");
}
```

`start()`: This method is called **each time** the applet is started (when the browser window is opened or refreshed).This is used for **starting threads** or other operations that need to be done when the applet is visible to the user. `public void start() {`

```
     // Start tasks when applet is
displayed
 System.out.println("Applet Started");
}
```

`paint()`: This method is called whenever the applet needs to **repaint** or redraw its interface.It is used for **rendering graphics** and **displaying information** on the applet window.

```
public void paint(Graphics g) {
        // Draw text or graphics
        g.drawString("Hello, Applet!", 50,
    50);
    }
```

`stop()`: This method is called when the applet is no longer visible (when the user navigates away from the page or the browser window is minimized).You can use it for tasks like **pausing threads** or **releasing resources**. `public void stop() {`

```
     // Stop tasks when applet is no
longer visible
     System.out.println("Applet Stopped");
    }
```

`destroy()`: This method is called when the applet is being **destroyed** (i.e., the user closes the browser or the applet is removed from the browser).Used for **clean-up** tasks like closing resources or terminating background threads.

```
public void destroy() {
        // Clean up resources when applet
    is destroyed
        System.out.println("Applet
    Destroyed");
    }
```

✓@@@ Simple Applet Display Methods

In addition to the life cycle methods, an applet can use a variety of **display methods** to draw and display content in the applet's window.

`paint()` Method

The `paint()` method is the most important method when displaying content in an applet. It is automatically called when the applet is first displayed, when it is resized, or when the user requests a repaint.

Example of the `paint()` method: `import java.applet.Applet;`

```
import java.awt.Graphics;

public class HelloWorldApplet extends Applet
{
    public void paint(Graphics g) {
        g.drawString("Hello, World!", 50,
50);  // Display "Hello, World!" at
coordinates (50, 50)
    }
}
```

`Graphics` Class Methods

You can use the `Graphics` class to draw shapes, lines, and text in an applet. Some common methods from the `Graphics` class include: **`drawString(String str, int x, int y)`**: Draws a string at the specified (x, y) coordinates. **`drawRect(int x, int y, int width, int height)`**: Draws a rectangle. **`fillRect(int x, int y, int width, int height)`**: Fills a rectangle with a color. **`setColor(Color c)`**: Sets the drawing color for subsequent drawing operations.

Example: Drawing Shapes in an Applet: `import java.applet.Applet;`

```
import java.awt.Graphics;
import java.awt.Color;

public class DrawingApplet extends Applet {
```

```
    public void paint(Graphics g) {
        g.setColor(Color.RED);
        g.fillRect(20, 20, 100, 50);  // Draw
a filled rectangle

        g.setColor(Color.BLUE);
        g.drawString("Applet Drawing!", 50,
100);  // Draw some text
    }
}
```

✓Example of a Simple Applet

Here's an example of a simple Java applet that displays a message on the screen: `import java.applet.Applet;`

```
import java.awt.Graphics;

public class SimpleApplet extends Applet {
    public void init() {
        // Initialization (called once)
        System.out.println("Applet
Initialized");
    }

    public void start() {
        // Start (called when applet becomes
visible)
        System.out.println("Applet Started");
    }

    public void paint(Graphics g) {
        // Display content
        g.drawString("Hello from Java
Applet!", 50, 50);
    }

    public void stop() {
        // Stop (called when applet is no
longer visible)
        System.out.println("Applet Stopped");
    }

    public void destroy() {
        // Cleanup (called before applet is
destroyed)
        System.out.println("Applet
Destroyed");
    }
}
```

✓Summary of Applet Life Cycle and Methods`init()`**: Initialization method, called once when the applet is loaded.**

1. **`start()`**: Called when the applet becomes visible.
2. **`paint()`**: Called to display graphics or text in the applet window.
3. **`stop()`**: Called when the applet is no longer visible.
4. **`destroy()`**: Cleanup method called when the applet is removed.

These methods allow you to control the behavior of an applet in response to user interaction or when the applet's state changes (visible, hidden, etc.).

Note: Modern Alternatives to Applets

As of recent years, applets are **deprecated** and no longer supported by modern browsers due to security concerns. Java **Applets** have been replaced by more modern approaches for creating interactive web applications such as **JavaFX**, **HTML5**, **JavaScript**, and **CSS**.

Would you like more examples or deeper explanations about applet graphics, or would you like to explore modern alternatives like JavaFX for GUI development?

✓Requesting Repainting in Java Applets

Repainting an applet means refreshing its graphical content. In Java applets, **repainting** is crucial when the content needs to be updated (for example, after a user interaction or a state change).

Repainting Using the `repaint()` Method

In Java, the `repaint()` method is used to request the applet to **redraw** its display. This method calls the `paint()` method of the applet, which is responsible for displaying the graphics. `import java.applet.Applet;`

```
import java.awt.Graphics;

public class RepaintingApplet extends Applet
{
    private int count = 0;

    // This method is called whenever
repaint() is called
    public void paint(Graphics g) {
        g.drawString("Count: " + count, 50,
50);
    }

    // This method is triggered on a button
click or some event
    public void incrementCount() {
        count++;   // Increment count
        repaint(); // Request repainting of
the applet to update the display
    }
}
```

`repaint()`: Requests that the applet be repainted, causing the `paint()` method to be called again.**`paint(Graphics g)`**: The method that gets invoked to draw content onto the applet's window. You can use the `Graphics` object to draw shapes, text, or images.

When should you use repaint()?

You should call `repaint()` whenever you want the applet to be refreshed or redrawn, typically in response to user actions (like pressing a button, moving the mouse, etc.).

✓Using the Status Window

In Java applets, a **status window** can be used to show status information, messages, or errors. It can be helpful for providing feedback to the user, such as displaying a message when an applet is loaded or when an error occurs.The **status window** is managed by the browser or the applet viewer. The method `showStatus(String message)` is used to display messages in the browser's status bar.

Example: Using `showStatus()`:

```
import java.applet.Applet;
import java.awt.Graphics;

public class StatusWindowApplet extends
Applet {
    public void init() {
```

```
        // Show a status message when the
applet initializes
        showStatus("Applet Initialized");
    }

    public void paint(Graphics g) {
        // Show a status message while
painting the applet
        showStatus("Displaying content...");
        g.drawString("Welcome to the
Applet!", 50, 50);
    }

    public void stop() {
        // Clear status message when the
applet is no longer visible
        showStatus("Applet Stopped");
    }
}
```

Key Points:**`showStatus(String message)`** : Displays the `message` in the status bar of the browser.It is particularly useful for showing progress, error messages, or general status information during the execution of an applet.

✓The HTML `<applet>` Tag

The `<applet>` tag in HTML is used to embed a Java applet into a webpage. This tag points to the applet's class file and allows you to specify the width, height, and other parameters for the applet's display area.

Syntax of the `<applet>` Tag:

```
<applet code="AppletClassName.class"
width="300" height="300">
    Your browser does not support Java
applets.
</applet>
```

- **`code`**: Specifies the name of the applet class file (with `.class` extension).
- **`width` and `height`**: Set the dimensions of the applet's display area.

Example of Embedding an Applet:

```
<applet code="SimpleApplet.class" width="300"
height="200">
    Your browser does not support Java
applets.
</applet>
```

In this case, the `SimpleApplet.class` will be loaded and displayed with a width of 300 pixels and a height of 200 pixels.

✓Passing Parameters to Applets

You can pass parameters to applets using the `<param>` tag within the `<applet>` tag. These parameters can be retrieved inside the applet to customize its behavior.

Using the `<param>` Tag:

```
<applet code="ParamApplet.class" width="300"
height="200">
    <param name="message" value="Hello,
World!">
    <param name="color" value="red">
</applet>
```

- **`<param>`**: This tag is used to pass parameters to the applet.
- **`name`**: The parameter's name.
- **`value`**: The value associated with the parameter.

The applet can then retrieve these parameters using the `getParameter()` method.

Example: Retrieving Parameters in the Applet

```
import java.applet.Applet;
import java.awt.Graphics;

public class ParamApplet extends Applet {
    private String message;
    private String color;

    public void init() {
        // Retrieve parameters passed from
the HTML file
        message = getParameter("message");
        color = getParameter("color");
    }

    public void paint(Graphics g) {
        // Set the color based on the passed
parameter
        if (color.equals("red")) {
            g.setColor(java.awt.Color.RED);
        } else {
            g.setColor(java.awt.Color.BLACK);
        }

        // Display the message
        g.drawString(message, 50, 50);
    }
}
```

Unit:- 4

@@@@@✓ AWT (Abstract Window Toolkit) in Java

The **Abstract Window Toolkit (AWT)** is Java's original platform-independent windowing toolkit, which provides a set of APIs for creating graphical user interfaces (GUIs) such as windows, buttons, text fields, and menus. AWT provides the basic functionality for creating and managing user interfaces in Java applications. It is less flexible compared to newer frameworks like **Swing** and **JavaFX**, but it is still used for basic GUI programming.

✓Working with AWT Controls

AWT provides a variety of **controls** (also called **widgets** or **components**) that allow interaction between the user and the application. Some common AWT controls include: **Button**: A clickable button.**TextField**: A single-line text input field.**Label**: A non-editable text label.**TextArea**: A multi-line text input field.**CheckBox**: A box that can either be checked or unchecked. **RadioButton**: A button in a group of radio buttons where only one button in the group can be selected at a time. **List**: A list of items that the user can select from. **ComboBox**: A drop-down list of options. **Scrollbar**: A scrollable area for large content.

Example of Creating and Using AWT Controls:

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

public class AWTControlsExample extends Frame
{
    Button button;
    TextField textField;
```

```
    AWTControlsExample() {
        // Creating a button and adding an
action listener
        button = new Button("Click Me");
        button.setBounds(100, 100, 80, 30);
        button.addActionListener(new
ActionListener() {
            public void
actionPerformed(ActionEvent e) {
                textField.setText("Button
clicked!");
            }
        });

        // Creating a text field
        textField = new TextField();
        textField.setBounds(50, 150, 200,
30);

        // Adding controls to the frame
        add(button);
        add(textField);

        // Setting layout and frame
properties
        setLayout(null);
        setSize(300, 300);
        setVisible(true);
    }

    public static void main(String[] args) {
        new AWTControlsExample(); // Create
the frame with controls
    }
}
```

✓ AWT Classes

AWT provides several important **classes** that are used for creating and managing GUI components and handling events. **`Frame`**: Represents a top-level window with a title and a border. **`Button`**: Represents a clickable button. **`TextField`**: A single-line text field. **`Label`**: Displays a text message. **`Panel`**: A container for organizing components. **`Dialog`**: A pop-up window used for communication with the user. **`Scrollbar`**: Adds scrollbars for large content. **`List`**: Represents a list of items. Some of the common **layout managers** in AWT include: **FlowLayout**: Arranges components in a left-to-right flow. **BorderLayout**: Divides the container into five regions: north, south, east, west, and center. **GridLayout**: Organizes components into a grid of rows and columns.

Example of Using a Layout Manager:

```
import java.awt.*;

public class AWTLayoutExample extends Frame {
    AWTLayoutExample() {
        // Using BorderLayout
        setLayout(new BorderLayout());

        Button button1 = new Button("Button
1");
        Button button2 = new Button("Button
2");

        add(button1, BorderLayout.NORTH);
        add(button2, BorderLayout.SOUTH);

        setSize(300, 200);
        setVisible(true);
    }
    public static void main(String[] args) {
        new AWTLayoutExample(); // Create the
frame with layout
    }
}
```

✓ Window Fundamentals in AWT

A **window** in AWT is typically created using the **`Frame`** class. A `Frame` is a top-level container that represents a window with a title bar, close button, and borders. Key components of an AWT window: **Title Bar**: Displays the window's title. **Close Button**: Typically located at the top-right corner of the window. It allows users to close the window. **Content Area**: The area within the window where components are displayed. **Menu Bar**: An optional bar that can contain menus like File, Edit, etc.

Example of Creating a Basic Window with AWT:

```
import java.awt.*;

public class BasicWindowExample {
    public static void main(String[] args) {
        // Creating a Frame (a top-level
window)
        Frame frame = new Frame("AWT Window
Example");

        // Setting window size and visibility
        frame.setSize(400, 300);
        frame.setVisible(true);

        // Close the window when the user
clicks the close button
        frame.addWindowListener(new
java.awt.event.WindowAdapter() {
            public void
windowClosing(java.awt.event.WindowEvent we)
{
                System.exit(0);
            }
        });
    }
}
```

✓ Working with Frames

The **`Frame`** class in AWT is used for creating a top-level window with a title and border. The `Frame` class is a subclass of the `Window` class and provides all the basic functionality to create windows in AWT.

Key Methods of the `Frame` Class:

- **`setTitle(String title)`**: Sets the title of the frame.
- **`setSize(int width, int height)`**: Sets the size of the window.
- **`setVisible(boolean visibility)`**: Controls whether the frame is visible or not.
- **`setLayout(LayoutManager manager)`**: Sets the layout of the frame (how components are arranged inside the frame).

✓ Creating a Frame Window in an Applet

Although applets are primarily used to display content inside a browser window, it is possible to create a **frame window** within an applet by using the `Frame` class.

Example of Creating a Frame in an Applet:

```
import java.applet.Applet;
import java.awt.*;

public class FrameInApplet extends Applet {
    public void init() {
        // Create a frame inside the applet
        Frame frame = new Frame("Applet with
Frame");

        // Setting frame size and making it
visible
        frame.setSize(400, 300);
        frame.setVisible(true);

        // Add a button inside the frame
        Button button = new Button("Click
Me");
        frame.add(button);

        // Add a window listener to close the
frame
        frame.addWindowListener(new
java.awt.event.WindowAdapter() {
            public void
windowClosing(java.awt.event.WindowEvent we)
{
                System.exit(0);
            }
        });
    }
}
```

@@@@@ ✓ Displaying Information Within a Window

Displaying information within a window is done using various AWT components like **`Label`**, **`TextField`**, **`TextArea`**, **`Button`**, and others. The `paint()` method of a `Frame` can also be used to display custom content such as text or graphics.

Example of Displaying Information in a Window:

```
import java.awt.*;

public class DisplayInfoWindow {
    public static void main(String[] args) {
        Frame frame = new Frame("Information
Display");

        // Creating a Label to display
information
        Label label = new Label("Welcome to
AWT!");
        label.setBounds(50, 50, 200, 30);

        // Add the label to the frame
        frame.add(label);

        // Setting the layout and size
        frame.setSize(300, 200);
        frame.setLayout(null);  // No layout
manager, manually placing components
        frame.setVisible(true);

        // Add a window listener to close the
frame
        frame.addWindowListener(new
java.awt.event.WindowAdapter() {
            public void
windowClosing(java.awt.event.WindowEvent we)
{
                System.exit(0);
            }
        });
    }
} In this example:A Label component is
created and added to the frame.The
setBounds() method is used to specify the
position and size of the label inside the
frame.
```

@@@@ ✓Working with Graphics in Java AWT

In Java AWT (Abstract Window Toolkit), **graphics** are used to display images, shapes, and text on a window. You can create graphical content using the `Graphics` class, which provides various methods to draw basic shapes, text, and images.

Graphics Class in Java

The `Graphics` class is responsible for drawing on a component (like a `Frame` or `Panel`). The most common operations are done in the `paint()` method, which is called whenever the component needs to be redrawn.

Here are some common methods used in the `Graphics` class:
`drawLine(int x1, int y1, int x2, int y2)`: Draws a line between the coordinates `(x1, y1)` and `(x2, y2)`.
`drawRect(int x, int y, int width, int height)`: Draws a rectangle at the given position with the specified width and height. **`fillRect(int x, int y, int width, int height)`**: Draws a filled rectangle.
`drawOval(int x, int y, int width, int height)`: Draws an oval inside the specified bounding rectangle.
`fillOval(int x, int y, int width, int height)`: Draws a filled oval.

Example: Working with Graphics

```
import java.awt.*;

public class GraphicsExample extends Frame {
    public void paint(Graphics g) {
        // Drawing basic shapes
        g.drawLine(50, 50, 150, 150); // Line
        g.drawRect(50, 200, 100, 50); //
Rectangle
        g.fillRect(200, 200, 100, 50); //
Filled Rectangle
        g.drawOval(50, 300, 100, 50); // Oval
        g.fillOval(200, 300, 100, 50); //
Filled Oval

        // Drawing text
        g.setColor(Color.BLUE); // Set color
to blue
        g.drawString("Hello, AWT Graphics!",
50, 400);
    }

    public static void main(String[] args) {
        GraphicsExample ge = new
GraphicsExample();
        ge.setSize(400, 500);
        ge.setVisible(true);
    }
}
```

✓Working with Color

In AWT, colors are managed using the `Color` class. This class provides predefined color constants like `Color.RED`,

`Color.GREEN`, and `Color.BLUE`, but you can also create custom colors using RGB values.

Setting Color in Graphics

To set the color of graphics, you use the `setColor()` method of the `Graphics` class. Here's how to change the color of shapes and text: **Predefined Colors**: `Color.RED`, `Color.BLUE`, `Color.GREEN`, etc. **Custom Colors**: You can define a custom color by specifying RGB values using `new Color(r, g, b)`.

Example: Working with Color

```
import java.awt.*;

public class ColorExample extends Frame {
    public void paint(Graphics g) {
        // Set color for drawing
        g.setColor(Color.RED); // Red color
        g.fillRect(50, 50, 100, 100); //
Filled red rectangle

        g.setColor(new Color(0, 255, 0)); //
Custom green color
        g.fillOval(200, 50, 100, 100); //
Filled green oval

        g.setColor(Color.BLUE); // Blue color
        g.drawString("Color Example", 50,
200); // Blue text
    }

    public static void main(String[] args) {
        ColorExample ce = new ColorExample();
        ce.setSize(400, 300);
        ce.setVisible(true);
    }
}
```

✓Setting the Paint Mode

The **paint mode** in Java determines how a new shape or drawing operation affects the existing drawing. By default, when you draw a shape, it simply overwrites what was there before. However, you can set the "**XOR mode**" which means the drawing is performed in a way that it "inverts" the underlying content, effectively "erasing" the previous drawing.

Setting Paint Mode Example

```
import java.awt.*;

public class PaintModeExample extends Frame {
    public void paint(Graphics g) {
        // Set XOR paint mode
        g.setXORMode(Color.YELLOW); // XOR
mode with yellow color
        g.drawRect(50, 50, 100, 100); // XOR
the rectangle with yellow
    }

    public static void main(String[] args) {
        PaintModeExample pm = new
PaintModeExample();
        pm.setSize(400, 300);
        pm.setVisible(true);
    }
}
```

`setXORMode(Color c)`: Sets the XOR mode for drawing. The color passed to this method will be used in the XOR operation. XOR mode is typically used for effects like highlighting, and it is not as commonly used as standard drawing operations.

✓Working with Fonts

AWT allows you to customize the font style, size, and family when drawing text using the `Graphics` class. The `Font` class is used to represent font attributes.

Setting Fonts

You can create a custom font and set it using the `setFont()` method of the `Graphics` class.

Example: Working with Fonts

```
import java.awt.*;

public class FontExample extends Frame {
    public void paint(Graphics g) {
        // Create a custom font
        Font font = new Font("Arial",
Font.BOLD, 20);
        g.setFont(font); // Set the font

        // Draw some text with the custom
font
        g.setColor(Color.BLACK);
        g.drawString("Custom Font Example",
50, 100);
    }

    public static void main(String[] args) {
        FontExample fe = new FontExample();
        fe.setSize(400, 200);
        fe.setVisible(true);
    }
}
```

Font constructor: `Font(String name, int style, int size)`

`name`: The font family (e.g., "Arial", "Times New Roman"). `style`: The style of the font, such as `Font.BOLD` or `Font.ITALIC`. `size`: The size of the font. **`setFont(Font f)`**: This method sets the font to be used for drawing text.

✓Exploring Text and Graphics

AWT provides a variety of options for both **drawing text** and **displaying graphics**. The `Graphics` class has several methods for working with text: **`drawString(String str, int x, int y)`**: Draws the text at the specified `(x, y)` coordinates. **`drawChars(char[] data, int offset, int length, int x, int y)`**: Draws a string of characters starting at the specified offset.

You can also combine graphics and text to create complex GUI elements, such as buttons or labels with custom designs.

Example: Combining Text and Graphics

```
import java.awt.*;

public class TextAndGraphicsExample extends
Frame {
    public void paint(Graphics g) {
        // Draw text
        g.setColor(Color.BLUE);
        g.setFont(new Font("Serif",
Font.BOLD, 24));
        g.drawString("Text and Graphics
Example", 50, 50);

        // Draw a shape
```

```
        g.setColor(Color.RED);
        g.fillRect(50, 100, 100, 50); // A
filled rectangle
    }

    public static void main(String[] args) {
        TextAndGraphicsExample tg = new
TextAndGraphicsExample();
        tg.setSize(400, 200);
        tg.setVisible(true);
    }
}
```

✅Layout Managers and Menus in Java AWT

Layout Managers in AWT

A **layout manager** controls the size and position of components within a container (such as a `Frame` or `Panel`). AWT provides several types of layout managers: **FlowLayout**: Components are arranged in left-to-right flow. **BorderLayout**: Divides the container into five regions: North, South, East, West, and Center. **GridLayout**: Arranges components in a grid of rows and columns. **CardLayout**: Manages multiple components (cards) and allows you to switch between them.

Example: Using FlowLayout

```
import java.awt.*;

public class FlowLayoutExample extends Frame
{
    public FlowLayoutExample() {
        setLayout(new FlowLayout()); // Set
FlowLayout for arranging components

        Button button1 = new Button("Button
1");
        Button button2 = new Button("Button
2");

        add(button1);
        add(button2);

        setSize(300, 200);
        setVisible(true);
    }

    public static void main(String[] args) {
        new FlowLayoutExample(); // Create
and display the frame
    }
}
```

Menus in AWT

AWT provides the `MenuBar`, `Menu`, and `MenuItem` classes to create menus in applications. You can create menus like File, Edit, etc., and add items that the user can click.

Example: Creating a Simple Menu

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

public class MenuExample extends Frame {
    public MenuExample() {
        MenuBar menuBar = new MenuBar();

        // Create File menu
        Menu fileMenu = new Menu("File");
        MenuItem newItem = new
MenuItem("New");
        MenuItem exitItem = new
MenuItem("Exit");


 fileMenu.add(newItem);
        fileMenu.add(exitItem);

        // Add file menu to menu bar
        menuBar.add(fileMenu);
        setMenuBar(menuBar);

        // Add action listener to the
Exit menu item
        exitItem.addActionListener(new
ActionListener() {
            public void
actionPerformed(ActionEvent e) {
                System.exit(0);
            }
        });

        setSize(300, 200);
        setVisible(true);
    }

    public static void main(String[]
args) {
        new MenuExample(); // Create and
display the window
    }
}
```