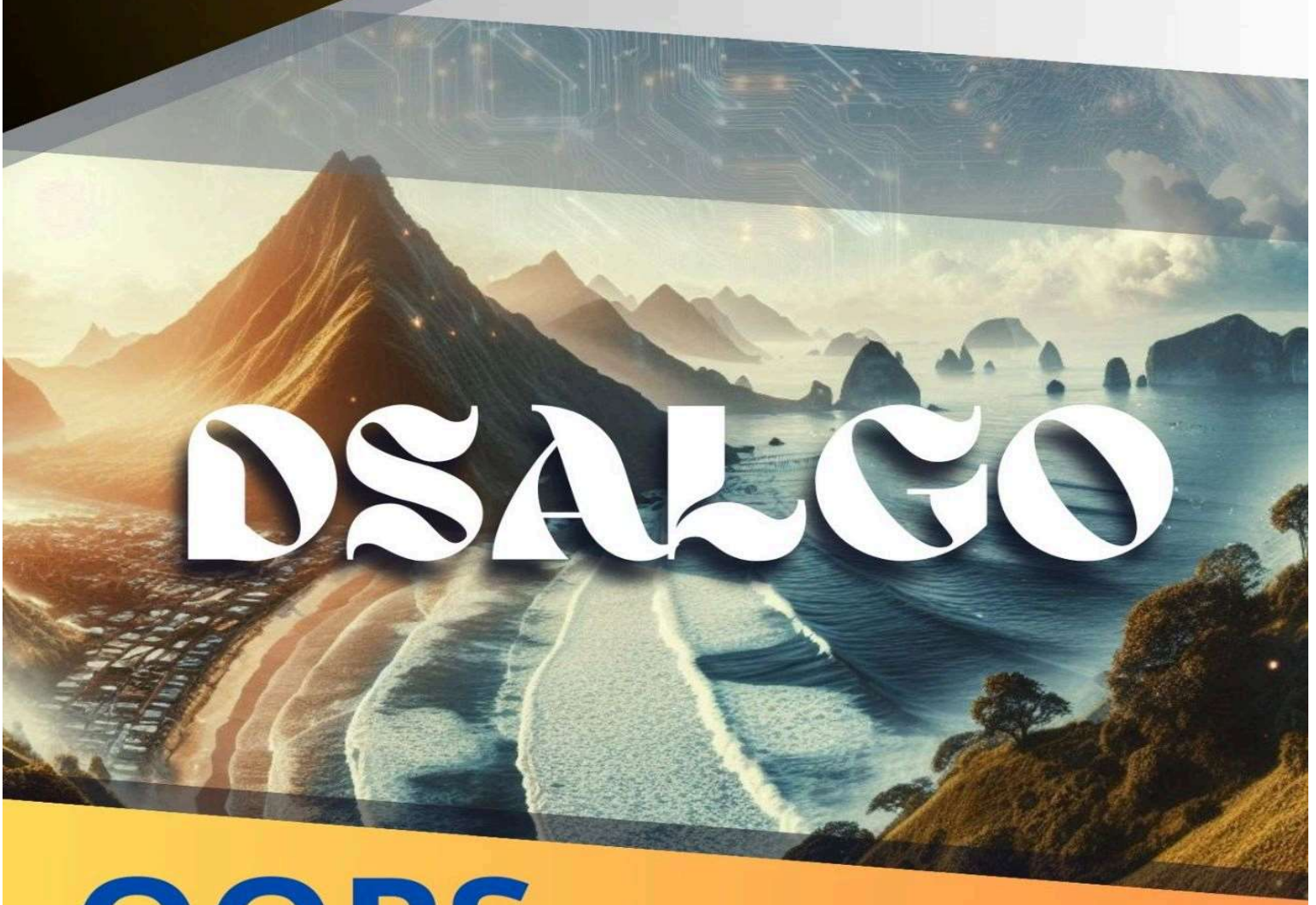
E-BOOK

UNIT -1

PDF



OOPS WITH JAVA

written By : Abhay Kumar Singh

1.1 Why Java ? / introduction of JAVA

- Java is a high-level, object-oriented programming language developed by Sun Microsystems in the mid-1990s.
- It is designed to be platform-independent, making it an ideal choice for cross-platform applications.
- Key features include:
  1) **Portability**: Write Once, Run Anywhere (WORA) capability.
  2) **Object-Oriented**: Encourages modular and reusable code.
  3) **Robustness**: Strong memory management and exception handling.
  4) **Security**: Provides a secure runtime environment.
  5) **Performance**: Efficient execution with Just-In-Time (JIT) compilation.
  6) **Multithreading**: Built-in support for concurrent programming.

1.2 History of Java

- **1991**: Initiated as the "Green Project" by James Gosling, Mike Sheridan, and Patrick Naughton.
- **1995**: Officially released as Java 1.0 by Sun Microsystems.
- **1996**: First major release, Java Development Kit (JDK) 1.0.
- **1997**: Introduction of Java 1.1 with inner classes and JavaBeans.
- **2004**: Java 5.0 introduced generics, metadata, and enumerated types.
- **2009**: Oracle acquires Sun Microsystems, continues Java development.
- **2017**: Introduction of Java 9 with module system.
- **Present**: Regular feature releases every six months.

1.3 JVM (Java Virtual Machine)

- The JVM is the runtime engine that executes Java bytecode. Key responsibilities include:
  1) **Bytecode Execution**: Converts Java bytecode into machine code.
  2) **Memory Management:** Handles allocation and garbage collection.
- **Platform Independence**: Provides a uniform execution environment across different platforms.

1.4 JRE (Java Runtime Environment)

- The JRE is a package that provides the necessary libraries, JVM, and other components to run Java applications.
- It includes:
  1) **Java Libraries:** Standard libraries needed for running Java programs.
  2) **Java Virtual Machine**: The interpreter to execute Java bytecode.
  3) **Runtime Tools:** Utilities like Java Web Start and Java Plugin.

1.5 Java Environment

- The Java environment consists of the following components:
  1) **JDK (Java Development Kit)**: Includes JRE and development tools like the compiler (javac), debugger, and other tools necessary for Java development.
  2) **JRE (Java Runtime Environment)**: Provides libraries and JVM for running applications.
  3) **JVM (Java Virtual Machine):** Executes the bytecode.

1.6 Java Source File Structure

- A typical Java source file follows a specific structure:
  1) **Package Declaration (optional)**: Defines the package to which the class belongs
  2) **Import Statements (optional)**: Imports other Java classes or packages.
  3) **Class Declaration**: Defines the class and its members.

Filename.java

```java
package com.example.myapp;
import java.util.List;
public class MyClass {
    // Fields, constructors, methods
}
```

1.7 Compilation Fundamentals

- Java source code is compiled into bytecode using the Java compiler (javac)
- the bytecode is executed by the JVM.
- The compilation process involves:
  1) **Writing Source Code**: Create a .java file

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

  2) **Compiling:** Convert source code into bytecode.

```
javac HelloWorld.java
```

  3) **Running:** Execute the bytecode with the JVM

```
java HelloWorld
```

1.8 Defining Classes in Java

- A class in Java is a blueprint for creating objects.
- It encapsulates data for the object and methods to manipulate that data.

```java
public class MyClass {
    // Fields
    int number;
    String text;

    // Methods
    void display() {
        System.out.println("Number: " + number + ", Text: " + text);
    }
}
```

1.9 Constructors

- Constructors are special methods used to initialize objects. They have the same name as the class and no return type.

```java
public class MyClass {
    int number;
    String text;

    // Constructor
    public MyClass(int number, String text) {
        this.number = number;
        this.text = text;
    }
}
```

1.10 Methods

- Methods define the behavior of the objects created from the class. They can take parameters and return a value.

```java
public class MyClass {
    int number;
    String text;
    void display() {
        System.out.println("Number: " + number + ", Text: " + text);
    }


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

1.11 Access Specifiers

Access specifiers determine the visibility of class members. Java provides four types of access specifiers:

- **public**: Accessible from any other class.
- **protected**: Accessible within the same package and subclasses.
- **default** (no modifier): Accessible only within the same package.
- **private**: Accessible only within the same class.

```
public class MyClass {
   public int number;
   protected String text;
   int defaultAccess;
   private boolean isPrivate;
}
```

1.12 Final Members

The `final` keyword is used to declare constants and prevent inheritance or method overriding.

```
public class MyClass {
   final int constantNumber = 10;
   final void display() {
      System.out.println("This method cannot be overridden");
   }
}
```

1.13 Comments

- Comments are used to explain code and are ignored by the compiler.
  1) **Single-line**
  ```
  // This is a single-line comment
  ```
  2) **Multi-line**
  ```
  /* This is a
     multi-line comment */
  ```
  3) **Javadoc:**
  ```
  /**
   * This is a Javadoc comment
   * used to generate documentation.
   */
  ```

1.14 Data Types

- Java provides several data types categorized into two types:
  1) **Primitive**: byte, short, int, long, float, double, char, boolean.
  2) **Reference**: Arrays, Classes, Interfaces.

```
int number = 10;
float decimal = 3.14f;
char letter = 'A';
boolean flag = true;
```

1.15 Variables

- Variables are containers for storing data values.

```
dtype var_name1; // define
var_name1=4;   // intialize
dtype var_name2=5; // define + initialize
```

1.16 Types of variables:

- **Local Variables**: Declared inside a method.

```
public class LocalVariableExample {
   public void display() {
      int localVar = 10; // Local variable
      System.out.println("Local Variable: " + localVar);
   }

   public static void main(String[] args) {
      LocalVariableExample obj = new LocalVariableExample();
      obj.display();
   }
}
```

- **Instance Variables**: Declared in a class, but outside any method.

```
public class InsVar {
   int value ; // Instance variable

   public void display() {
      System.out.println("Instance Variable: " + value);
   }

   public static void main(String[] args) {
      InsVar obj1 = new InsVar();
      obj1.value = 5;
      obj1.display();
      InsVar obj2 = new InsVar();
      obj2.value = 10;
      obj2.display();
   }
}
```

- **Static Variables**: Declared with the static keyword and inside the class but outside any method or constructor .

```
public class StVar {
   static int value; // Static variable

   public void display() {
      System.out.println("Static Variable: " + value);
   }

   public static void main(String[] args) {
      StVar obj1 = new StVar();
      StVar.value = 15; // Accessing static variable through class name
      obj1.display();

      StVar obj2 = new StVar();
      StVar.value = 30; // Accessing static variable through class name
      obj2.display();

      obj1.display(); // Static variable value is updated for all instances
   }
}
```

Program1 : write a program to add two integers.
Program2: WAP to calculate average of three numbers

1.17 Operators

- Operators are special symbols that perform operations on variables and values. Java has several types of operators:
  1) **Arithmetic Operators:** +, -, *, /, %
  2) **Relational Operators:** ==, !=, >, <, >=, <=
  3) **Logical Operators:** &&, ||, !
  4) **Assignment Operators:** =, +=, -=, *=, /=, %=

```
int a = 10, b = 20;
int sum = a + b; // Arithmetic
boolean result = (a < b); // Relational
boolean logical = (a > 5) && (b > 15); // Logical
```

1.18 Control Flow Statements in Java

1. if, else if, else: Conditional Statements

- These statements allow the execution of certain blocks of code based on specified conditions.
- **Syntax**:

```
if (condition) {
   // code to be executed if condition is true  }
else if (anotherCondition) {
  // code to be executed if anotherCondition is true  }
else {
   // code to be executed if all conditions are false  }
```

Example:

```
public class IfElseExample {
   public static void main(String[] args) {
      int number = 10;

      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.");
      }
   }
}
```

Output:
The number is positive.

2. switch: A Multi-way Branch Statement

- The switch statement allows a variable to be tested for equality against a list of values.
- **Syntax**:

```
switch (variable) {
   case value1:
      // code to be executed if variable == value1
            break;
   case value2:
      // code to be executed if variable == value2
      break;
            // more cases
   default:
      // code to be executed if variable doesn't match any case
}
```

Example:

```
public class SwitchExample {
   public static void main(String[] args) {
      int n= 3;
      String dayName;
      switch (n) {
         case 1:
            result= "case 1 executed !";
            break;
         case 2:
            result= "case 2 executed !";
            break;
         default:
            result = "other case executed!";
            break;
      }
      System.out.println("The day is " + result);
   }  }
```

1.19 Looping Statements

- These loops repeatedly execute a block of code as long as the condition remains true.

1.while Loop

Syntax:

```
while (condition) {
   // code to be executed
}
```

Example:

```
public class WhileExample {
   public static void main(String[] args) {
      int count = 1;
      while (count <= 5) {
         System.out.println("Count is: " + count);
         count++;
      }
   }
}
```

```
Output :
Count is: 1
Count is: 2
Count is: 3
Count is: 4
Count is: 5
```

2.do-while Loop

- **Syntax:**

```
do {
   // code to be executed
} while (condition);
```

Example:

```
public class DoWhileExample {
   public static void main(String[] args) {
      int count = 1;
      do {
         System.out.println("Count is: " + count);
         count++;
      } while (count <= 5);
   } }
```

Output :
Count is: 1
Count is: 2
Count is: 3
Count is: 4
Count is: 5

3. for: A Compact Loop Statement

- The for loop is commonly used when the number of iterations is known beforehand.

Syntax:

```
for (initialization; condition; increment/decrement) {
   // code to be executed
}
```

Example:

```
public class ForExample {
   public static void main(String[] args) {
      for (int i = 1; i <= 5; i++) {
         System.out.println("Count is: " + i);
      }
   }
}
```

Output:
Count is: 1
Count is: 2
Count is: 3
Count is: 4
Count is: 5

1.20 Control Loop Execution

- These statements control the flow of loops, allowing early termination or skipping iterations.

1.break Statement

- Terminates the loop immediately.

2.continue Statement

- Skips the current iteration and proceeds with the next iteration.

```
public class ContinueExample {
   public static void main(String[] args) {
      for (int i = 1; i <= 5; i++) {  // for break keyword
         if (i == 3) {
            break;
         }
         System.out.println("Count is: " + i);
      }
```

```
    for (int i = 1; i <= 5; i++) {  // for continue keyword
        if (i == 3) {
            continue;
        }
        System.out.println("Count is: " + i);
    }
 }
}
```

Output :

```
Count is: 1
Count is: 2
Count is: 1
Count is: 2
Count is: 4
Count is: 5
```

Program : WAP to check number is odd or even

1.21 Arrays

- Arrays are used to store multiple values of the same type in a single variable.
- Instead of declaring separate variables for each value, you can store all the values in a single array variable.
- Once an array is created, its size cannot be changed.
- Array indexing starts at 0.
- All elements in an array are of the same type.
- Elements are stored in contiguous memory locations.
- **Syntax**:

```
dataType[] arrayName;  //method 1 : defining

dataType arrayName[];  //method 2 : defining

arrayName = new dataType[size];  // initializing

dataType[] arrayName = new dataType[size];   //: defining & initializing

dataType[] arrayName = {n1,n2,n3,.....,nm};   //: defining & initializing value

arrayName[index]; //accessing the value
```

Example of Array

```
public class ArrayExample {
   public static void main(String[] args) {
      int[] numbers = {1, 2, 3, 4, 5};
       System.out.println("index 1 " +  ": " + numbers[1]);

      // Using for loop
      for (int i = 0; i < numbers.length; i++) {
         System.out.println("Element at index " + i + ": " + numbers[i]);
      }  } }
```

Output:

```
index 1: 2
Element at index 0: 1
Element at index 1: 2
Element at index 2: 3
Element at index 3: 4
Element at index 4: 5
```

1.22 Multidimensional Arrays

- Java supports multidimensional arrays, primarily two-dimensional arrays (arrays of arrays).
- **Syntax**:

```
dataType[][] arrayName = new dataType[rows][columns];
```

Example:

```
int[][] matrix = new int[3][3];
int[][] predefinedMatrix = {
   {1, 2, 3},
   {4, 5, 6},
   {7, 8, 9}
};
```

Program: WAP to calculate the sum of the array .
Program: WAP to Multiply two matrices.
Program : WAP to add Two Matrices.

1.23 Strings

- A string in Java is a sequence of characters.
- Strings are immutable, meaning once a string is created, it cannot be changed. Any modification creates a new string.
- Java maintains a pool of strings to optimize memory usage and improve performance.

- **Creating Strings**

```
String str1 = "Hello, World!";
String str2 = new String("Hello, World!");
```

1.24 Common String Methods

- **length**: Returns the length of the string.
- **charAt**: Returns the character at the specified index.
- **substring**: Returns a substring from the specified start index to the end or up to the specified end index.
- **equals**: Compares two strings for equality.
- **toLowerCase and toUpperCase:** Converts all characters of the string to lower or upper case.
- **replace**: Replaces all occurrences of a specified character or sequence with another character or sequence.
- **split**: Splits the string into an array of substrings based on the specified delimiter.

1.25 Object-Oriented Programming (OOP)

- It is a programming paradigm based on the concept of "objects," which are instances of classes.
- OOP principles in Java help in creating modular, reusable, and maintainable code

1.26 Class

- A class is a blueprint for creating objects.
- It defines a datatype by bundling data and methods that work on the data.

Example:

```
public class Animal {
   // Fields
   String name;
   int age;

   // Constructor
   public Animal(String name, int age) {
      this.name = name;
      this.age = age;
   }

// Method
   public void makeSound() {
      System.out.println("Some sound...");
   }
}
```

1.27 Object

- An object is an instance of a class. It has state (attributes/fields) and behavior (methods).

Example:

```
public class Main {
   public static void main(String[] args) {
      Animal dog = new Animal("Buddy", 3);
      dog.makeSound(); // Output: Some sound...
   }
}
```

1.28 Inheritance

- Inheritance allows a new class to inherit the properties and methods of an existing class.
- Super Class (Parent Class) The class being inherited from.
- Sub Class (Child Class) The class that inherits from another class.

Example:

```
// Super Class
public class Animal {
   public void makeSound() {
      System.out.println("Some sound...");
   }
}

// Sub Class
public class Dog extends Animal {
    public void makeSound() {
      System.out.println("Bark");
   }
}
```

1.29 Overriding

- Overriding occurs when a subclass provides a specific implementation of a method already defined in its superclass.

Example:

```
public class Dog extends Animal {
   @Override
   public void makeSound() {
      System.out.println("Bark");
   }
}
```

1.30 Overloading

- Overloading occurs when two or more methods in the same class have the same name but different parameters.

Example:

```
public class MathUtil {
   public int add(int a, int b) {
      return a + b;
   }
   public int add(int a, int b, int c) {
      return a + b + c;
   }
}
```

1.31 Encapsulation

- Encapsulation is the mechanism of wrapping the data (variables) and the code acting on the data (methods) together as a single unit.

Example:

```
public class Person {
   private String name;
   private int age;

   public String getName() {
      return name;
   }
   public void setName(String name) {
      this.name = name;
   }
   public int getAge() {
      return age;
   }
   public void setAge(int age) {
      this.age = age;
   }
}
```

1.32 Polymorphism

- Polymorphism allows objects to be treated as instances of their parent class rather than their actual class.

Example:

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

1.33 Abstraction

- Abstraction is the concept of hiding the complex implementation details and showing only the necessary features of an object.
- **Abstract Class** : A class that cannot be instantiated and is meant to be subclassed.

Example:

```
public abstract class Animal {
   public abstract void makeSound();
}

// Subclass
public class Dog extends Animal {
   public void makeSound() {
      System.out.println("Bark");
   }
}
```

1.34 Interfaces

- An interface is a reference type in Java. It is similar to a class and is a collection of abstract methods.

Example:

```
public interface Animal {
   void makeSound();
}

public class Dog implements Animal {
   public void makeSound() {
      System.out.println("Bark");
   }
}
```

1.35 Example Program Demonstrating OOP

```
// Abstract class
abstract class Animal {
   String name;

   Animal(String name) {
      this.name = name;
   }
   abstract void makeSound();
}

// Interface
interface Pet {
   void play();
}

// Subclass
class Dog extends Animal implements Pet {
   Dog(String name) {
      super(name);
   }

   @Override
   void makeSound() {
      System.out.println(name + " says: Bark");
   }

   @Override
   public void play() {
      System.out.println(name + " loves to play fetch!");
   }
}
```

```
// Main class
public class Main {
   public static void main(String[] args) {
      Dog myDog = new Dog("Buddy");
      myDog.makeSound();
      myDog.play();
   }
}
Output :
Buddy says: Bark
Buddy loves to play fetch!
```

1.36 Defining a Package

- To define a package, use the package keyword followed by the package name at the beginning of a Java source file.

Example:

```
package com.example.myapp;

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

1.37 CLASSPATH Setting for Packages

- The CLASSPATH is an environment variable that tells the Java Virtual Machine (JVM) and Java compiler where to look for user-defined classes and packages.

```
java -cp /path/to/classes com.example.myapp.MyClass
```

1.38 Making JAR Files for Library Packages

- JAR (Java Archive) files bundle multiple files into a single archive file, typically used to distribute Java classes and associated metadata.

i.Creating a JAR file:

1.Compile your Java classes:

```
javac com/example/myapp/*.java
```

2.Create the JAR file:

```
jar cf myapp.jar com/example/myapp/*.class
```

ii.Add a manifest file (optional) to specify the main class:

```
echo "Main-Class: com.example.myapp.MainClass" > manifest.txt

jar cfm myapp.jar manifest.txt com/example/myapp/*.class
```

iii.Using the JAR file:

```
java -cp myapp.jar com.example.myapp.MyClass
```

1.39 Import

- The import statement allows you to use classes from other packages without needing to specify the full package name.

Example:

```
import java.util.Date;

public class Test {
   public static void main(String[] args) {
      Date date = new Date();
      System.out.println(date);
   } }
```

1.40 Naming Convention for Packages

- Package names in Java typically follow a hierarchical naming pattern,
- starting with the top-level domain name in reverse, followed by the organization and project names.

Example:

- Top-level domain: com, org, net
- Organization name: example
- Project name: myapp

Resulting package name: `com.example.myapp`

1.41 Static Import

- To use static import, you use the import static statement followed by the fully qualified name of the static member you want to import.
- You can also use a wildcard (*) to import all static members of a class.
- **Syntax**

i.Single Static Member Import:

```
import static packageName.ClassName.staticMember;
```

ii.Import All Static Members:

```
import static packageName.ClassName.*;
```