OBJECT ORIENTED PROGRAMMING USING JAVA

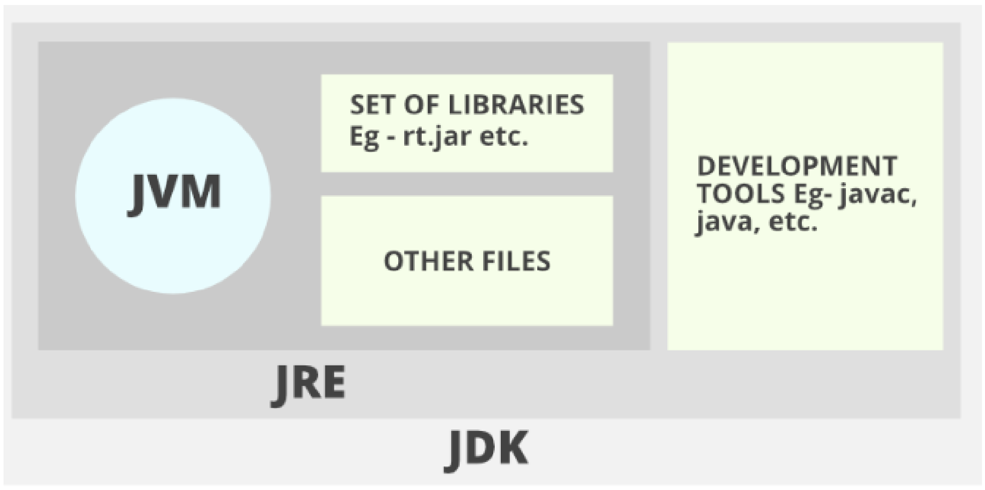
MODULE – II

Introduction to java applications – Introduction to classes, objects, methods & Strings - Control statements – Arrays - constructor – function overloading & overriding - Inheritance - Polymorphism – Interface – package - exception handling

Introduction to java applications

Setting up the environment in Java

Java is a general-purpose computer programming language that is concurrent, class-based, object-oriented, etc. Java applications are typically compiled to bytecode that can run on any Java virtual machine (JVM) regardless of computer architecture. The latest version is **Java 19**. Below are the environment settings for both Linux and Windows. JVM, JRE, and JDK three are all platform-dependent because the configuration of each Operating System is different.


JVM
SET OF LIBRARIES Eg - rt.jar etc.
OTHER FILES
DEVELOPMENT TOOLS Eg- javac, java, etc.
JRE
JDK

- JDK(Java Development Kit): JDK is intended for software developers and includes development tools such as the Java compiler, Javadoc, Jar, and a debugger.
- JRE(Java Runtime Environment): JRE contains the parts of the Java libraries required to run Java programs and is intended for end-users. JRE can be viewed as a subset of JDK.
- JVM: JVM (Java Virtual Machine) is an abstract machine. It is a specification that provides a runtime environment in which java bytecode can be executed.

JVMs are available for many hardware and software platforms.

Java program is a collection of objects, and these objects communicate through method calls to each other to work together. Here is a brief discussion on the Classes and Objects, Method, Instance variables, syntax, and semantics of Java.

Basic terminologies in Java

- **class** keyword is used to declare a class in Java.
- **public** keyword is an access modifier that represents visibility. It means it is visible to all.
- **static** is a keyword. If we declare any method as static, it is known as the static method. The core advantage of the static method is that there is no need to create an object to invoke the static method.
- **void** is the return type of the method. It means it doesn't return any value.
- **main** represents the starting point of the program.
- **String args[]** is used for command line argument. We will discuss it in coming section.
- **System.out.println()** is used to print statement.

```
class Simple{
    public static void main(String args[]){
     System.out.println("Hello Java");
    }
}
```

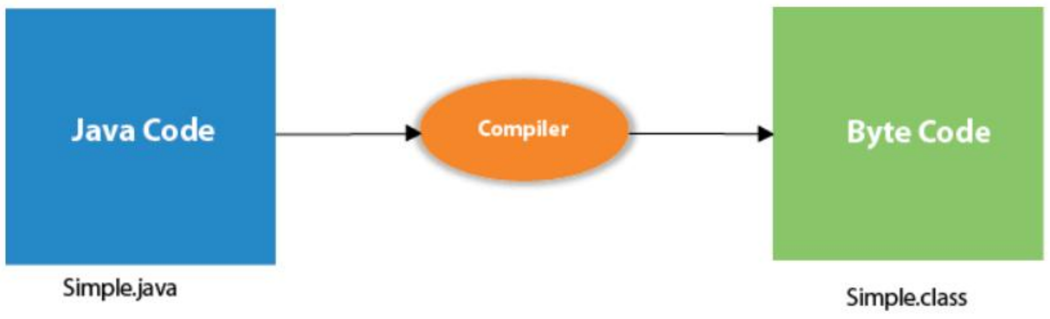
Save the above file as Simple.java.

To compile: javac Simple.java

To execute: java Simple

Output

```
Hello Java
```



Constructors

In Java, a constructor is similar to the method, It should be in class name. We can call the constructor when an instance of the class is created. Memory for the object is allocated at the time of calling constructor.

It is a special type of method which is used to initialize the object.

Every time an object is created using the new() keyword, at least one constructor is called.

It calls a default constructor if there is no constructor available in the class. In such case, Java compiler provides a default constructor by default.

Types of constructors

Default Constructor / Empty Constructor

Parameterized Constructor

Constructor Overloading

Example: Default Constructor

```
class Bike{
        String type;
        Bike(){
                System.out.println("Bike is created");
        }
        public static void main(String args[]){
                Bike b1=new Bike();
        }
}
```

In the above example program Bike(){} is the default constructor, It will call at the time of b1 instance memory allocation.

Example: Parameterized Constructor

```
class Student{
   int id;
   String name;
   Student(int i, String n){
   id = i;
   name = n;
   }
   void display(){
        System.out.println(id+" "+name);
        }
  public static void main(String args[]){
   Student s1 = new Student(111,"Karan");
   Student s2 = new Student(222,"Aryan");
   s1.display();
   s2.display();
  }
}
```

In the above example **Student(int i, String n)** is the parameterized constructor, we can send data to the newly created object through this type of constructor.

Methods

We used functions in C and C++ programming. But in java It is called methods.

There are two types of method creation in java.

1. Static method

   This type of method we can call without instance variable.

2. Non static method

   We must use instance variable at the time of calling non static methods. However it is the member of the object.

Creating Method

Method definition consists of a method header and a method body. The same is shown in the following syntax −

General Syntax

```
modifier returnType nameOfMethod (Parameter List) {
   // method body
}
```

The syntax shown above includes −

modifier − It defines the access type of the method and it is optional to use.

returnType − Method may return a value.

nameOfMethod − This is the method name. The method signature consists of the method name and the parameter list.

Parameter List − The list of parameters, it is the type, order, and number of parameters of a method. These are optional, method may contain zero parameters.

method body − The method body defines what the method does with the statements.

Considering the following example to explain the syntax of a method −

Syntax

```
public static int methodName(int a, int b) {
  // body
}
```

Here,

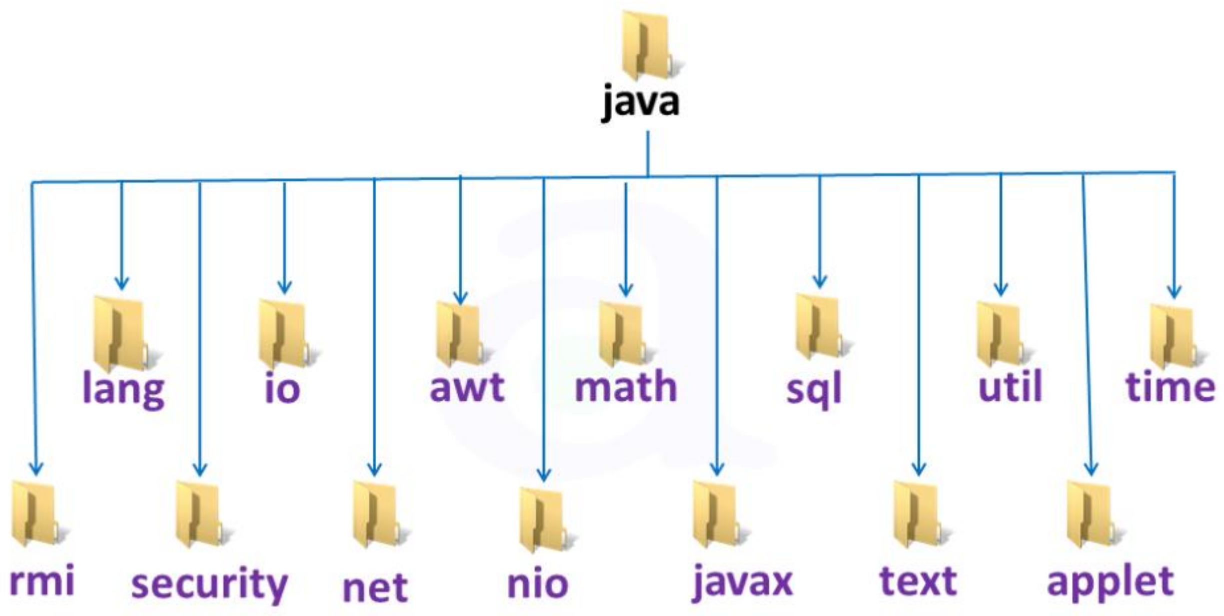
- public static − modifier
- int − return type
- methodName − name of the method
- int a, int b − list of parameters

Packages in Java

Java package is a group of similar types of classes, interfaces and sub-packages or we can say Packages in Java is a mechanism to encapsulate a group of classes, interfaces and sub packages which is used to providing access protection and namespace management and to make searching/locating and usage of classes, interfaces, enumerations and annotations easier.

Package in Java can be categorized in two form, **built-in package**, and **user-defined package**.

There are many built-in packages such as java, lang, awt,io, util, javax, swing, net, sql etc. Here, we will have the detailed learning of creating and using user-defined packages.



The library is divided into packages and classes. Meaning you can either import a single class (along with its methods and attributes), or a whole package that contain all the classes that belong to the specified package.

To use a class or a package from the library, you need to use the import keyword:

Syntax

```
import package.name.Class;      // Import a single class
import package.name.*;          // Import the whole package
```

The following Example statement is show how to import object from package.

```
import java.util.Scanner;
```

In the example above, java.util is a package, while Scanner is a class of the java.util package. To use the Scanner class, create an object of the class and use any of the available methods found in the Scanner class documentation. In our example, we will use the nextLine() method, which is used to read a complete line:

Example

Using the Scanner class to get user input:

```
import java.util.Scanner;
class MyClass {
  public static void main(String[] args) {
    Scanner myObj = new Scanner(System.in);
    System.out.println("Enter username");

    String userName = myObj.nextLine();
    System.out.println("Username is: " + userName);
  }
}
```

Advantage of Java Package

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.
2) Java package provides access protection.
3) In real life situation there may arise scenarios where we need to define files of the same name. This may lead to name-space collisions. Java package removes naming collision.
4) Reusability: Reusability of code is one of the most important requirements in the software industry. Reusability saves time, effort and also ensures consistency. A class once developed can be reused by any number of programs wishing to incorporate the class in that particular program.
5) Easy to locate the files.

Exception Handling

When executing Java code, different errors can occur: coding errors made by the programmer, errors due to wrong input, or other unforeseeable things.
When an error occurs, Java will normally stop and generate an error message. The technical term for this is: Java will throw an exception (throw an error).

Java try and catch

The try statement allows you to define a block of code to be tested for errors while it is being executed. The catch statement allows you to define a block of code to be executed, if an error occurs in the try block.
The try and catch keywords come in pairs:

Syntax

```
try {
  //  Block of code to try
}
catch(Exception e) {
  //  Block of code to handle errors
}
```

Consider the following example:

This will generate an error, because myNumbers[10] does not exist.

```
public class Main {
  public static void main(String[ ] args) {
    int[] myNumbers = {1, 2, 3};
    System.out.println(myNumbers[10]); // error!
  }
}
```

The output will be something like this:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 10
        at Main.main(Main.java:4)
```

If an error occurs, we can use try...catch to catch the error and execute some code to handle it:

Example

```
public class Main {
  public static void main(String[ ] args) {
    try {
      int[] myNumbers = {1, 2, 3};
      System.out.println(myNumbers[10]);
    } catch (Exception e) {
      System.out.println("Array subscript error.");
    }
```

```
        System.out.println(“Thank you”);
  }
}
```

The following output shows Thank you output is to continuous execution of the program.

The output will be:

```
Array subscript error.
Thank you
```

Polymorphism / Overloading

The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.

Method Overloading: When there are multiple functions with the same name but different parameters then these functions are said to be overloaded. Functions can be overloaded by change in the number of arguments or/and a change in the type of arguments.

```
class Helper
{
        void add(int a,int b)
{
        int z;
        z=a+b;
        System.out.println(z);
}
        void add(int a,int b,int c)
{
        int z;
        z=a+b+c;
        System.out.println(z);
}
void add(int a,int b,int c,int d)
{
        int z;
        z=a+b+c+d;
        System.out.println(z);
}
public static void main(String s[])
{
        Helper a=new Helper();
        a.add(12,34);
        a.add(56,67,34);
        a.add(2,4,5,6);
}
}
```

In the above example shows add() method overloaded with different number of arguments.

```
class Helper2
{
        void add(int a,int b)
{
        int z;
        z=a+b;
        System.out.println(z);
}
        void add(double a,double b)
{
        double z;
        z=a+b;
        System.out.println(z);
}

public static void main(String s[])
{
        Helper2 a=new Helper2();
        a.add(12,34);
        a.add(12.256, 13.456);}
}
```

The above Helper2 class shows add() method can be overloaded with different types arguments.

Overwriting

In the previous chapter, we talked about superclasses and subclasses. If a class inherits a method from its superclass, then there is a chance to override the method provided that it is not marked final.

The benefit of overriding is: ability to define a behavior that's specific to the subclass type, which means a subclass can implement a parent class method based on its requirement.

In object-oriented terms, overriding means to override the functionality of an existing method.
Example
Let us look at an example.

```
class Animal {
   public void move() {
      System.out.println("Animals can move");
   }
}

class Dog extends Animal {
   public void move() {
      System.out.println("Dogs can walk and run");
   }
}

public class TestDog {

   public static void main(String args[]) {
      Animal a = new Animal();   // Animal reference and object
      Animal b = new Dog();   // Animal reference but Dog object

      a.move();   // runs the method in Animal class
      b.move();   // runs the method in Dog class
   }
}
```

This will produce the following result −

Output
```
Animals can move
Dogs can walk and run
```

Control Statements

If is one of the control statement
Use the if statement to specify a block of Java code to be executed if a condition is true.

Syntax
```
if (condition) {
  // block of code to be executed if the condition is true
}
```
Note that if is in lowercase letters. Uppercase letters (If or IF) will generate an error.

In the example below, we test two values to find out if 20 is greater than 18. If the condition is true, print some text:

Example

```
if (20 > 18) {
  System.out.println("20 is greater than 18");
}
```

Use the else statement to specify a block of code to be executed if the condition is false.

Syntax

```
if (condition) {
  // block of code to be executed if the condition is true
} else {
  // block of code to be executed if the condition is false
}
```

Example

```
int time = 20;
if (time < 18) {
  System.out.println("Good day.");
} else {
  System.out.println("Good evening.");
}
// Outputs "Good evening."
```

Looping Statement

A loop statement allows us to execute a statement or group of statements multiple times and following is the general form of a loop statement in most of the programming languages −

Java programming language provides the following types of loop to handle looping requirements. Click the following links to check their detail.

1 while loop

Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.

Syntax

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

In the example below, the code in the loop will run, over and over again, as long as a variable (i) is less than 5:

Example

```
int i = 0;
while (i < 5) {
  System.out.println(i);
  i++;
}
```

2 for loop

Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable.

Syntax

```
for (statement 1; statement 2; statement 3) {
        // code block to be executed
}
```

Statement 1 is executed (one time) before the execution of the code block.

Statement 2 defines the condition for executing the code block.

Statement 3 is executed (every time) after the code block has been executed.
The example below will print the numbers 0 to 4:

Example

```
for (int i = 0; i < 5; i++) {
  System.out.println(i);
}
```

Statement 1 sets a variable before the loop starts (int i = 0).

Statement 2 defines the condition for the loop to run (i must be less than 5). If the condition is true, the loop will start over again, if it is false, the loop will end.

Statement 3 increases a value (i++) each time the code block in the loop has been executed.

3 do...while loop

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

Syntax

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

The example below uses a do/while loop. The loop will always be executed at least once, even if the condition is false, because the code block is executed before the condition is tested:

Example

```
int i = 0;
do {
  System.out.println(i);
  i++;
}
while (i < 5);
```

Java supports the following control statements.

Sr.No. Control Statement & Description

1 break statement

Terminates the loop or switch statement and transfers execution to the statement immediately following the loop or switch.

This example stops the loop when i is equal to 4:

Example

```
for (int i = 0; i < 10; i++) {
  if (i == 4) {
    break;
  }
  System.out.println(i);
}
```

2 continue statement

Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.

This example skips the value of 4:

Example

```
for (int i = 0; i < 10; i++) {
  if (i == 4) {
    continue;
  }
  System.out.println(i);
}
```

Array

Java provides a data structure, the array, which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

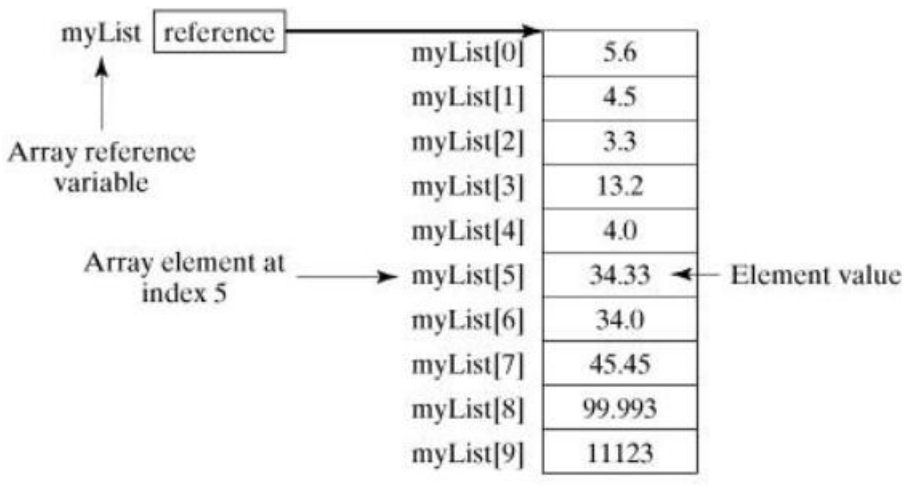
Declaring an array variable, creating an array, and assigning the reference of the array to the variable can be combined in one statement, as shown below −

dataType[] arrayRefVar = new dataType[arraySize];
Alternatively you can create arrays as follows −

dataType[] arrayRefVar = {value0, value1, ..., valuek};
The array elements are accessed through the index. Array indices are 0-based; that is, they start from 0 to arrayRefVar.length-1.

Example
Following statement declares an array variable, myList, creates an array of 10 elements of double type and assigns its reference to myList −

double[] myList = new double[10];
Following picture represents array myList. Here, myList holds ten double values and the indices are from 0 to 9.



```
public class TestArray {
  public static void main(String[] args) {
    double[] myList = {1.9, 2.9, 3.4, 3.5};
    // Print all the array elements
    for (int i = 0; i < myList.length; i++) {
      System.out.println(myList[i] + " ");
    }
    // Summing all elements
    double total = 0;
    for (int i = 0; i < myList.length; i++) {
      total += myList[i];
    }
    System.out.println("Total is " + total);

    // Finding the largest element
    double max = myList[0];
```

```
      for (int i = 1; i < myList.length; i++) {
         if (myList[i] > max) max = myList[i];
      }
      System.out.println("Max is " + max);
   }
}
```

This will produce the following result −

```
Output
1.9
2.9
3.4
3.5
Total is 11.7
Max is 3.5
```

Interfaces

An interface is a reference type in Java. It is similar to class. It is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface.

Along with abstract methods, an interface may also contain constants, default methods, static methods, and nested types. Method bodies exist only for default methods and static methods.

Writing an interface is similar to writing a class. But a class describes the attributes and behaviors of an object. And an interface contains behaviors that a class implements.

Unless the class that implements the interface is abstract, all the methods of the interface need to be defined in the class.

An interface is similar to a class in the following ways −

An interface can contain any number of methods.

An interface is written in a file with a .java extension, with the name of the interface matching the name of the file.

The byte code of an interface appears in a .class file.

Interfaces appear in packages, and their corresponding bytecode file must be in a directory structure that matches the package name.

However, an interface is different from a class in several ways, including −

You cannot instantiate an interface.

An interface does not contain any constructors.

All of the methods in an interface are abstract.

An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.

An interface is not extended by a class; it is implemented by a class.

An interface can extend multiple interfaces.

Declaring Interfaces
The interface keyword is used to declare an interface. Here is a simple example to declare an interface −

Example
Following is an example of an interface −

```
/* File name : NameOfInterface.java */
import java.lang.*;
// Any number of import statements

public interface NameOfInterface {
  // Any number of final, static fields
  // Any number of abstract method declarations\
}
```

Interfaces have the following properties −

An interface is implicitly abstract. You do not need to use the abstract keyword while declaring an interface.

Each method in an interface is also implicitly abstract, so the abstract keyword is not needed.

Methods in an interface are implicitly public.

Example

```
/* File name : Animal.java */
interface Animal {
  public void eat();
  public void travel();
}
```

Implementing Interfaces
When a class implements an interface, you can think of the class as signing a contract,

agreeing to perform the specific behaviors of the interface. If a class does not perform all the behaviors of the interface, the class must declare itself as abstract.

A class uses the implements keyword to implement an interface. The implements keyword appears in the class declaration following the extends portion of the declaration.

Example

```
/* File name : Mammal.java */
public class Mammal implements Animal {

   public void eat() {
      System.out.println("Mammal eats");
   }

   public void travel() {
      System.out.println("Mammal travels");
   }

   public int noOfLegs() {
      return 0;
   }

   public static void main(String args[]) {
      Mammal m = new Mammal();
      m.eat();
      m.travel();
   }
}
```

This will produce the following result −

Output

```
Mammal eats
Mammal travels
```