Index

**Introduction to Java**

Java is a high-level programming language originally developed by Sun Microsystems , which is now the maintained by Oracle. Java was released in the year 1995. James Gosling is known as the father of Java.

Before Java it was named as Oak(Oak is a symbol of strength and chosen as a national tree of many countries). In 1995, Oak was renamed as "Java" because it was already a trademark by Oak Technologies. Java is an island in Indonesia where the first coffee was produced (called Java coffee). Java name was chosen by James Gosling.

**Java Applications**

1. standalone Application

2. Web Application

3. Enterprise Application

4. Mobile Application

JVM(Java Virtual Machine) acts as a run-time engine to run Java applications. JVM is the one that actually calls the main method present in Java code. JVM is a part of the JRE(Java Runtime Environment).

Java applications are called "Write Once Run Anywhere". This means a programmer can develop Java code on one system and can expect it to run on any other Java-enabled system without any adjustment. This is all possible because of JVM.

In Java, the program is not converted to code directly understood by Hardware, rather it is converted to bytecode(.class file), which is interpreted by JVM, so once compiled it generates bytecode file, which can be run anywhere (in any machine) which has JVM.

**Practical Implementation of WORA using a simple JAVA program to check whether a number is even or odd.**

import java.util.Scanner;

class GFG {

public static void main(String args[])

{

int num;

System.out.println("Enter a number:");

Scanner input = new Scanner(System.in);

num = input.nextInt();

if (num % 2 == 0)

System.out.println(num + " is even");

else

System.out.println(num + " is odd");

}

}

For Compiling (done on Windows 10):

javac GFG.java

After compilation there will be a class file in the corresponding folder named as:

GFG.class

When copied the bytecode (.class) generated on compilation to another platform(like mac os) and running it we get same output.

**output:**

Enter a number:

54

54 is even

Java, when compiled, creates a bytecode (.class file), which can be run in any machine which supports JVM. So once compiled it doesn’t require re-compilation at every machine it runs, JVM converts the bytecode to be understood by the underlying hardware.

The primary objective of Java programming language creation was to make it portable, simple and secure programming language. In spite of this, Java have number of feature that makes it a popular language of programming. The features of Java are also known as Java buzzwords.

**Features of the Java language is given below.**

Simple

Object-Oriented

Portable

Platform independent

Secured

Robust

Architecture neutral

Interpreted

High Performance

Multithreaded

Object-oriented means we organize our application as a combination of different type of objects that contains both data and behavior. Basic concepts of Object-Oriented Programming are:

Object

Class

Inheritance

Polymorphism

Abstraction

Encapsulation

**Simple**

Java is very easy to learn, and its syntax is simple, clean and easy to understand.

Java language is a simple programming language because:

* Java has removed many complicated and rarely-used features, for example, explicit pointers, operator overloading, etc.
* There is no need to remove unreferenced objects because there is an Automatic Garbage Collection in Java.

### Platform Independent

 A platform is the hardware or software environment in which a program runs. Java code can be executed on multiple platforms, for example, Windows, Linux, Sun Solaris, Mac/OS, etc. Java code is compiled by the compiler and converted into bytecode. This bytecode is a platform-independent code because it can be run on multiple platforms.

#### **Secure**

With Java's secure feature it enables to develop virus-free, tamper-free systems. Authentication techniques are based on public-key encryption.

Java is secured because:

* No explicit pointer
* Java Programs run inside a virtual machine sandbox

Java language provides these securities by default. Some security can also be provided by an application developer explicitly through SSL, JAAS, Cryptography, etc.

**Robust**

The English mining of Robust is strong. Java is robust because:

* It uses strong memory management.
* There is a lack of pointers that avoids security problems.
* Java provides automatic garbage collection which runs on the Java Virtual Machine to get rid of objects which are not being used by a Java application anymore.
* There are exception handling and the type checking mechanism in Java. All these points make Java robust.

#### **Architecture-neutral**

Java compiler generates an architecture-neutral object file format, which makes the compiled code executable on many processors, with the presence of Java runtime system.

#### **Portable**

Being architecture-neutral and having no implementation dependent aspects of the specification makes Java portable. The compiler in Java is written in ANSI C with a clean portability boundary, which is a POSIX subset.

#### **High Performance**

With the use of Just-In-Time compilers, Java enables high performance.

### Distributed

Java is distributed because it facilitates users to create distributed applications in Java. RMI and EJB are used for creating distributed applications. This feature of Java makes us able to access files by calling the methods from any machine on the internet.

### Multi-threaded

With Java's multithreaded feature it is possible to write programs that can perform many tasks simultaneously. The main advantage of multi-threading is that it doesn't occupy memory for each thread. It shares a common memory area. Threads are important for multi-media, Web applications, etc.

#### **Dynamic**

Java is considered to be more dynamic than C or C++ since it is designed to adapt to an evolving environment. Java programs can carry an extensive amount of run-time information that can be used to verify and resolve accesses to objects at run-time.

**Java Program Lifecycle**

Java programs normally undergo four phases

**Edit**

Programmer writes program (and stores program on disk)

**Compile**

Compiler creates byte codes from program (.class)

**Load**

Class loader stores byte codes in memory

**Execute**

Interpreter: translates byte codes into machine language

# **Wrapper Classes in Java**

A Wrapper class is a class whose object wraps or contains primitive data types. When we create an object to a wrapper class, it contains a field and in this field, we can store primitive data types. In other words, we can wrap a primitive value into a wrapper class object.

## **Autoboxing**

The automatic conversion of primitive data type into its corresponding wrapper class is known as autoboxing, for example, byte to Byte, char to Character, int to Integer, long to Long, float to Float, boolean to Boolean, double to Double, and short to Short.

## **Unboxing**

The automatic conversion of wrapper type into its corresponding primitive type is known as unboxing. It is the reverse process of autoboxing.

**Example:**

public class Main {

public static void main(String[] args) {

Integer myInt = 5;

Double myDouble = 5.99;

Character myChar = 'A';

System.out.println(myInt.intValue());

System.out.println(myDouble.doubleValue());

System.out.println(myChar.charValue());

}

}

**Output:**

5

5.99

A

**Datatypes**

Java is a statically-typed language. This means that all variables must be declared before they can be used.

Data types specify the different sizes and values that can be stored in the variable. There are two types of data types in Java:

1. **Primitive data types:** The primitive data types include boolean, char, byte, short, int, long, float and double.
2. **Non-primitive data types:** The non-primitive data types include [Classes, Interfaces, Arrays](https://www.javatpoint.com/object-and-class-in-java)

Primitive Data Types The Java programming language defines eight primitive types:

Logical

boolean

Textual

char

Integral

Byte

Short

Int

long

Floating

Double

float

A primitive data type specifies the size and type of variable values, and it has no additional methods.

## **Non-Primitive Data Types**

Non-primitive data types are called **reference types** because they refer to objects.

The **Reference Data Types** will contain a memory address of variable value because the reference types won’t store the variable value directly in memory. They are [**strings**](https://www.geeksforgeeks.org/strings-in-java/), [**objects**](https://www.geeksforgeeks.org/classes-objects-java/), [arrays](https://www.geeksforgeeks.org/arrays-in-java/), etc.

**OOPS in Java**

OOP stands for **Object-Oriented Programming**. **Object** means a real-world entity such as a pen, chair, table, computer, watch, etc.  object-oriented programming is about creating objects that contain both data and methods.

Object-oriented programming has several advantages:

* OOP is faster and easier to execute
* OOP provides a clear structure for the programs
* OOP helps to keep the Java code DRY "Don't Repeat Yourself", and makes the code easier to maintain, modify and debug
* OOP makes it possible to create full reusable applications with less code and shorter development time

There are Four fundamental concepts of OOPS:

**Encapsulation**

**Abstraction**

**Polymorphism**

**Inheritance**

**Encapsulation:**

Encapsulation in Java is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit. In encapsulation, the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class. Therefore, it is also known as **data hiding**.

To achieve encapsulation in Java −

* Declare the variables of a class as private.
* Provide public setter and getter methods to modify and view the variables values.

**Example of Encapsulation in Java**

Let's see the simple example of encapsulation that has only one field with its setter and getter methods.

*File: Student.java*

//A Java class which is a fully encapsulated class.

//It has a private data member and getter and setter methods.

**package** com.point;

**public** **class** Student{

//private data member

**private** String name;

//getter method for name

**public** String getName(){

**return** name;

}

//setter method for name

**public** **void** setName(String name){

**this**.name=name

}

}

*File: Test.java*

1. //A Java class to test the encapsulated class.
2. **package** com.point;
3. **class** Test{
4. **public** **static** **void** main(String[] args){
5. //creating instance of the encapsulated class
6. Student s=**new** Student();
7. //setting value in the name member
8. s.setName("vijay");
9. //getting value of the name member
10. System.out.println(s.getName());
11. }
12. }

Compile by: javac -d.Test.java

Run by: javacom.point.Test

**Output:**

Vijay

## **Advantages of Encapsulation**

* The fields of a class can be made read-only or write-only.
* A class can have total control over what is stored in its fields.

# **Abstraction in Java**

Data Abstraction is the property by virtue of which only the essential details are displayed to the user. The trivial or the non-essentials units are not displayed to the user. Ex: A car is viewed as a car rather than its individual components.

Data Abstraction may also be defined as the process of identifying only the required characteristics of an object ignoring the irrelevant details. The properties and behaviours of an object differentiate it from other objects of similar type and also help in classifying/grouping the objects.

Consider a real-life example of a man driving a car. The man only knows that pressing the accelerators will increase the speed of a car or applying brakes will stop the car, but he does not know about how on pressing the accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of the accelerator, brakes, etc in the car. This is what abstraction is.

In java, abstraction is achieved by [interfaces](https://www.geeksforgeeks.org/interfaces-in-java/) and [abstract classes](https://www.geeksforgeeks.org/abstract-classes-in-java/).

**Advantages of Abstraction**

1. It reduces the complexity of viewing the things.
2. Avoids code duplication and increases reusability.
3. Helps to increase the security of an application or program as only important details are provided to the user.

### Example of Abstract class in java

1. **abstract** **class** Bank{
2. **abstract** **int** getRateOfInterest();
3. }
4. **class** SBI **extends** Bank{
5. **int** getRateOfInterest(){**return** 7;}
6. }
7. **class** PNB **extends** Bank{
8. **int** getRateOfInterest(){**return** 8;}
9. }
11. **class** TestBank{
12. **public** **static** **void** main(String args[]){
13. Bank b;
14. b=**new** SBI();
15. System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
16. b=**new** PNB();
17. System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
18. }}

**Output:**

Rate of Interest is: 7%

Rate of Interest is: 8%

# **Polymorphism in Java**

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.

**Real life example of polymorphism:** A person at the same time can have different characteristic. Like a man at the same time is a father, a husband, an employee. So the same person posses different behavior in different situations. This is called polymorphism.

In other words, polymorphism allows you to define one interface and have multiple implementations. The word “poly” means many and “morphs” means forms, So it means many forms.

**In Java polymorphism is mainly divided into two types:**

* Compile time Polymorphism
* Runtime Polymorphism

1. **Compile-time polymorphism**: It is also known as static polymorphism. This type of polymorphism is achieved by function overloading or operator overloading. But **Java doesn’t support the Operator Overloading**.

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

**Example:**

**class** MultiplyFun {

    // Method with 2 parameter

**static** **int** Multiply(**int** a, **int** b)

    {

**return** a \* b;

    }

    // Method with the same name but 2 double parameter

**static** **double** Multiply(**double** a, **double** b)

    {

**return** a \* b;

    }

}

**class** Main {

**public** **static** **void** main(String[] args)

    {

        System.out.println(MultiplyFun.Multiply(2, 4));

        System.out.println(MultiplyFun.Multiply(5.5, 6.3));

    }

}

**Output:**

8

34.65

**2.**[Runtime polymorphism](https://www.geeksforgeeks.org/dynamic-method-dispatch-runtime-polymorphism-java/): It is also known as Dynamic Method Dispatch. It is a process in which a function call to the overridden method is resolved at Runtime. This type of polymorphism is achieved by Method Overriding.

[Method overriding](https://www.geeksforgeeks.org/overriding-in-java/), on the other hand, occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be **overridden**.

**Example:**

|  |
| --- |
| // Java program for Method overriding    **class** Parent {    **void** Print()      {          System.out.println("parent class");      }  }    **class** subclass1 **extends** Parent {    **void** Print()      {          System.out.println("subclass1");      }  }    **class** subclass2 **extends** Parent {    **void** Print()      {          System.out.println("subclass2");      }  }    **class** TestPolymorphism3 {  **public** **static** **void** main(String[] args)      {            Parent a; |

a = **new** subclass1();

        a.Print();

        a = **new** subclass2();

        a.Print();

    }

}

**Output:**

subclass1

subclass2

Inheritance can be defined as the process where one class acquires the properties (methods and fields) of another. With the use of inheritance the information is made manageable in a hierarchical order.

The class which inherits the properties of other is known as subclass (derived class, child class) and the class whose properties are inherited is known as superclass (base class, parent class).

## **extends Keyword**

**extends** is the keyword used to inherit the properties of a class. Following is the syntax of extends keyword.

**Types of Inheritance in Java**

Below are the different types of inheritance which are supported by Java.

* 1. **Single Inheritance:**In single inheritance, subclasses inherit the features of one superclass. In the image below, class A serves as a base class for the derived class B.

**Example:**

**import** java.io.\*;

**import** java.lang.\*;

**import** java.util.\*;

**class** one {

**public** **void** print\_geek()

    {

        System.out.println("Geeks");

    }

}

**class** two **extends** one {

**public** **void** print\_for() { System.out.println("For"); }

}

// Driver class

**public** **class** Main {

**public** **static** **void** main(String[] args)

    {

        two g = **new** two();

        g.print\_geek();

        g.print\_for();

        g.print\_geek();

    }

}

**Output:**

Geeks

For

Geeks

* 1. **Multilevel Inheritance:**In Multilevel Inheritance, a derived class will be inheriting a base class and as well as the derived class also act as the base class to other class. In the below image, class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C.

**Example:**

**import** java.io.\*;

**import** java.lang.\*;

**import** java.util.\*;

**class** one {

**public** **void** print\_geek()

    {

        System.out.println("Geeks");

    }

}

**class** two **extends** one {

**public** **void** print\_for() { System.out.println("For"); }

}

**class** three **extends** two {

**public** **void** print\_geek()

    {

        System.out.println("Geeks");

    }

}

// Drived class

**public** **class** Main {

**public** **static** **void** main(String[] args)

    {

        three g = **new** three();

        g.print\_geek();

        g.print\_for();

        g.print\_geek();

    }

}

**Output:**

Geeks

For

Geeks

* 1. **Hierarchical Inheritance:**In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one subclass. In the below image, class A serves as a base class for the derived class B, C and D.

**Example:**

**class** A {

**public** **void** print\_A() { System.out.println("Class A"); }

}

**class** B **extends** A {

**public** **void** print\_B() { System.out.println("Class B"); }

}

**class** C **extends** A {

**public** **void** print\_C() { System.out.println("Class C"); }

}

**class** D **extends** A {

**public** **void** print\_D() { System.out.println("Class D"); }

}

// Driver Class

**public** **class** Test {

**public** **static** **void** main(String[] args)

    {

        B obj\_B = **new** B();

        obj\_B.print\_A();

        obj\_B.print\_B();

        C obj\_C = **new** C();

        obj\_C.print\_A();

        obj\_C.print\_C();

        D obj\_D = **new** D();

        obj\_D.print\_A();

        obj\_D.print\_D();

}

}

**Output:**

Class A

Class B

Class A

Class C

Class A

Class D

* 1. [**Multiple Inheritance**](https://www.geeksforgeeks.org/java-and-multiple-inheritance/)**(Through Interfaces):**In Multiple inheritances, one class can have more than one superclass and inherit features from all parent classes. Java does **not** support [multiple inheritances](https://www.geeksforgeeks.org/java-and-multiple-inheritance/) with classes. In java, we can achieve multiple inheritances only through [Interfaces](https://www.geeksforgeeks.org/interfaces-in-java/). In the image below, Class C is derived from interface A and B.

**Example:**

**import** java.io.\*;

**import** java.lang.\*;

**import** java.util.\*;

**interface** one {

**public** **void** print\_geek();

}

**interface** two {

**public** **void** print\_for();

}

**interface** three **extends** one, two {

**public** **void** print\_geek();

}

**class** child **implements** three {

    @Override **public** **void** print\_geek()

    {

        System.out.println("Geeks");

    }

**public** **void** print\_for() { System.out.println("for"); }

}

// Drived class

**public** **class** Main {

**public** **static** **void** main(String[] args)

    {

        child c = **new** child();

        c.print\_geek();

        c.print\_for();

        c.print\_geek();

    }

}

**Output:**

Geeks

For

Geeks

* 1. **Hybrid Inheritance(Through Interfaces):**It is a mix of two or more of the above types of inheritance. Since java doesn’t support multiple inheritances with classes, hybrid inheritance is also not possible with classes. In java, we can achieve hybrid inheritance only through [Interfaces](https://www.geeksforgeeks.org/interfaces-in-java/).

## **IS-A Relationship**

IS-A is a way of saying: This object is a type of that object. Let us see how the **extends** keyword is used to achieve inheritance.

public class Animal {

}

public class Mammal extends Animal {

}

public class Reptile extends Animal {

}

public class Dog extends Mammal {

}

Now, based on the above example, in Object-Oriented terms, the following are true −

* Animal is the superclass of Mammal class.
* Animal is the superclass of Reptile class.
* Mammal and Reptile are subclasses of Animal class.
* Dog is the subclass of both Mammal and Animal classes.

Now, if we consider the IS-A relationship, we can say −

* Mammal IS-A Animal
* Reptile IS-A Animal
* Dog IS-A Mammal
* Hence: Dog IS-A Animal as well

## **HAS-A relationship**

These relationships are mainly based on the usage. This determines whether a certain class **HAS-A** certain thing. This relationship helps to reduce duplication of code as well as bugs.

Lets look into an example −

**Example**

public class Vehicle{}

public class Speed{}

public class Van extends Vehicle {

private Speed sp;

}

This shows that class Van HAS-A Speed. By having a separate class for Speed, we do not have to put the entire code that belongs to speed inside the Van class, which makes it possible to reuse the Speed class in multiple applications.

In Object-Oriented feature, the users do not need to bother about which object is doing the real work. To achieve this, the Van class hides the implementation details from the users of the Van class. So, basically what happens is the users would ask the Van class to do a certain action and the Van class will either do the work by itself or ask another class to perform the action.