**Inheritance in Java**

**Inheritance in Java** is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of [OOPs](https://www.javatpoint.com/java-oops-concepts)

The idea behind inheritance in Java is that you can create new [classes](https://www.javatpoint.com/object-and-class-in-java)

that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.

### **Advantages of Inheritance**

**Minimizing duplicate code:** Key benefits of Inheritance include minimizing the identical code as it allows sharing of the common code among other subclasses.

**Flexibility:** Inheritance makes the code flexible to change, as you will adjust only in one place, and the rest of the code will work smoothly.

**Overriding:** With the help of Inheritance, you can override the methods of the base class.

**Data Hiding:** The base class in Inheritance decides which data to be kept private, such that the derived class will not be able to alter it.

### **Disadvantages of Inheritance**

**No Independence:** One of the main disadvantages of Inheritance in Java is that two classes, both the base and inherited class, get tightly bounded by each other. In simple terms, Programmers can not use these classes independently of each other.

**Decreases Execution Speed:** Another con of Inheritance is that it decreases the execution speed because Inheritance execution takes time and effort.

**Refactoring the Code:** If the user deletes the Super Class, then they have to refactor it if they have used it.

**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/).

**Polymorphism**

### **Polymorphism in Java** can be defined as a task that can perform a single action in different ways. Polymorphism occurs when there is inheritance, i.e. there are many classes that are related to each other. Polymorphism in java allows us to use these inherited properties to perform different tasks. Thus, allowing us to achieve the same action in many different ways.

### **Real-Life Examples of Polymorphism**

An individual can have different relationships with different people. A woman can be a mother, a daughter, a sister, a friend, all at the same time, i.e. she performs other behaviours in different situations.

The human body has different organs. Every organ has a different function to perform; the heart is responsible for blood flow, lungs for breathing, brain for cognitive activity and kidneys for excretion. So we have a standard method function that performs differently depending upon the organ of the body.

#### **Polymorphism in Java Example**

A superclass named “Shapes” has a method “area()”. Subclasses of “Shapes” can be “Triangle”, “circle”, “Rectangle”, etc. Each subclass has its way of calculating area. Using Inheritance and Polymorphism, the subclasses can use the “area()” method to find the area’s formula for that shape.

|  |
| --- |
| **class** Shapes {  **public** **void** area() {      System.out.println("The formula for area of ");    }  }  **class** Triangle **extends** Shapes {  **public** **void** area() {      System.out.println("Triangle is ½ \* base \* height ");    }  }  **class** Circle **extends** Shapes {  **public** **void** area() {      System.out.println("Circle is 3.14 \* radius \* radius ");    }  }  **class** Main {  **public** **static** **void** main(String[] args) {      Shapes myShape = **new** Shapes();  // Create a Shapes object      Shapes myTriangle = **new** Triangle();  // Create a Triangle object      Shapes myCircle = **new** Circle();  // Create a Circle object      myShape.area();      myTriangle.area();      myShape.area();      myCircle.area();    }  } |

**Output:**

The formula for the area of Triangle is ½ \* base \* height  
The formula for the area of the Circle is 3.14 \* radius \* radius

## **Types of Polymorphism**

Polymorphism in Java can be performed by two different methods:

1. Method Overloading
2. Method Overriding

[**Method overloading**](https://www.mygreatlearning.com/blog/method-overloading-in-java/)is defined as a process that can create multiple methods of the same name in the same class, and all the methods work in different ways. Method overloading occurs when there is more than one method of the same name in the class.

#### **Example of Method Overloading in Java**

|  |
| --- |
| **class** Shapes {  **public** **void** area() {      System.out.println("Find area ");    }  **public** **void** area(**int** r) {      System.out.println("Circle area = "+3.14\*r\*r);    }    **public** **void** area(**double** b, **double** h) {      System.out.println("Triangle area="+0.5\*b\*h);    }  **public** **void** area(**int** l, **int** b) {      System.out.println("Rectangle area="+l\*b);    }      }    **class** Main {  **public** **static** **void** main(String[] args) {      Shapes myShape = **new** Shapes();  // Create a Shapes object        myShape.area();      myShape.area(5);      myShape.area(6.0,1.2);      myShape.area(6,2);    }  } |

**Output:**

Find area  
Circle area = 78.5  
Triangle area=3.60  
Rectangle area=12

Method overriding is defined as a process when the subclass or a child class has the same method as declared in the parent class.

### **Example of Method Overriding in Java**

class Vehicle{

//defining a method

void run(){System.out.println("Vehicle is moving");}

}

//Creating a child class

class Car2 extends Vehicle{

//defining the same method as in the parent class

void run(){System.out.println("car is running safely");}

public static void main(String args[]){

Car2 obj = new Car2();//creating object

obj.run();//calling method

}

}

**Output**:

Car is running safely

**Interface**

Like a class, an interface can have methods and variables, but the methods declared in an interface are by default abstract (only method signature, no body).

* Interfaces specify what a class must do and not how. It is the blueprint of the class.
* An Interface is about capabilities like a Player may be an interface and any class implementing Player must be able to (or must implement) move(). So it specifies a set of methods that the class has to implement.
* If a class implements an interface and does not provide method bodies for all functions specified in the interface, then the class must be declared abstract.
* A Java library example is, [Comparator Interface](https://www.geeksforgeeks.org/comparator-interface-java/). If a class implements this interface, then it can be used to sort a collection.

Java Interface also **represents the IS-A relationship**.

**Uses of Interface**

There are mainly three reasons to use interface. They are given below.

* It is used to achieve abstraction.
* By interface, we can support the functionality of multiple inheritance.
* It can be used to achieve loose coupling.

**Example:**

**interface** printable{

**void** print();

}

**class** A6 **implements** printable{

**public** **void** print(){System.out.println("Hello");}

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

A6 obj = **new** A6();

obj.print();

 }

}

**Output:**

Hello

## **Interface inheritance**

A class implements an interface, but one interface extends another interface.

Example:

**interface** Printable{

**void** print();

}

**interface** Showable **extends** Printable{

**void** show();

}

**class** TestInterface4 **implements** Showable{

**public** **void** print(){System.out.println("Hello");}

**public** **void** show(){System.out.println("Welcome");}

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

TestInterface4 obj = **new** TestInterface4();

obj.print();

obj.show();

 }

}

**Output:**

Hello

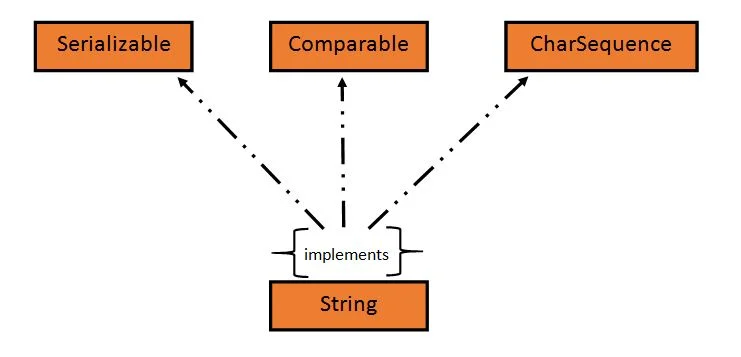
Welcome

**String Handling**

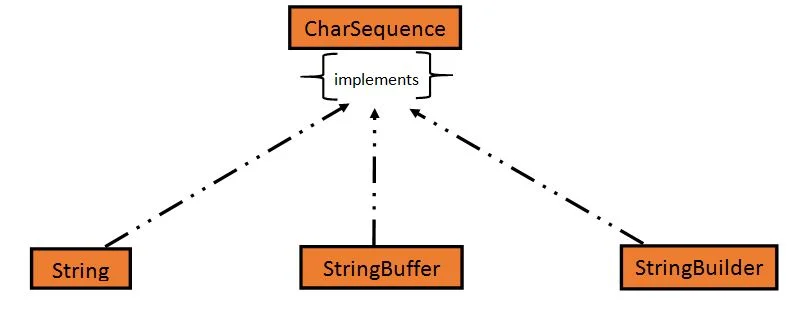
String is an **object** that represents sequence of characters. In Java, String is represented by String class which is located into java.lang package

It is probably the most commonly used class in java library. In java, every string that we create is actually an object of type **String**. One important thing to notice about string object is that string objects are **immutable** that means once a string object is created it cannot be changed.

The Java String class implements Serializable, Comparable and CharSequence interface that we have represented using the below image.



In Java, **CharSequence** Interface is used for representing a sequence of characters. CharSequence interface is implemented by String, StringBuffer and StringBuilder classes. This three classes can be used for creating strings in java.



### **Creating a String object**

String can be created in number of ways, here are a few ways of creating string object.

### **1) Using a String literal**

String literal is a simple string enclosed in double quotes " ". A string literal is treated as a String object.

public class Demo{

public static void main(String[] args) {

String s1 = "Hello Java";

System.out.println(s1);

}

**Output:**

Hello Java

### **2) Using new Keyword**

We can create a new string object by using **new** operator that allocates memory for the object.

public class Demo{

public static void main(String[] args) {

String s1 = new String("Hello Java");

System.out.println(s1);

}

}

**Output:**

Hello Java

### **Concatenating String**

There are 2 methods to concatenate two or more string.

1. Using **concat()** method
2. Using + operator

### **1) Using concat() method**

Concat() method is used to add two or more string into a single string object. It is string class method and returns a string object.

public class Demo{

public static void main(String[] args) {

String s = "Hello";

String str = "Java";

String str1 = s.concat(str);

System.out.println(str1);

}

}

**Output:**

HelloJava

### **2) Using + operator**

Java uses "+" operator to concatenate two string objects into single one. It can also concatenate numeric value with string object. See the below example.

public class Demo{

public static void main(String[] args) {

String s = "Hello";

String str = "Java";

String str1 = s+str;

String str2 = "Java"+11;

System.out.println(str1);

System.out.println(str2);

}

}

**Output:**

HelloJava

Java11

### **String Comparison**

To compare string objects, Java provides methods and operators both. So we can compare string in following three ways.

1. Using equals() method
2. Using == operator
3. By CompareTo() method

### **Using equals() method**

equals() method compares two strings for equality. Its general syntax is,

boolean equals (Object str)

### **Example**

It compares the content of the strings. It will return **true** if string matches, else returns **false**.

public class Demo{

public static void main(String[] args) {

String s = "Hell";

String s1 = "Hello";

String s2 = "Hello";

boolean b = s1.equals(s2); //true

System.out.println(b);

b = s.equals(s1) ; //false

System.out.println(b);

}

}

**Output:**

true

false

### **Using == operator**

The double equal (==) operator compares two object references to check whether they refer to same instance. This also, will return **true** on successful match else returns false.

public class Demo{

public static void main(String[] args) {

String s1 = "Java";

String s2 = "Java";

String s3 = new String ("Java");

boolean b = (s1 == s2); //true

System.out.println(b);

b = (s1 == s3); //false

System.out.println(b);

}

}

**Output:**

true

false

### **compareTo() method**

String compareTo() method compares values and returns an integer value which tells if the string compared is less than, equal to or greater than the other string. It compares the String based on natural ordering i.e alphabetically. Its general syntax is.

### **Syntax:**

int compareTo(String str)

**Example:**

### public class HelloWorld{

public static void main(String[] args) {

String s1 = "Abhi";

String s2 = "Viraaj";

String s3 = "Abhi";

int a = s1.compareTo(s2); //return -21 because s1 < s2

System.out.println(a);

a = s1.compareTo(s3); //return 0 because s1 == s3

System.out.println(a);

a = s2.compareTo(s1); //return 21 because s2 > s1

System.out.println(a);

}

}

**Output:**

-21

0

21