# *NOTES*

**(OOPS With Java)**

***Java OOPs Concepts***

====================================================================================

* Object-Oriented Programming is a paradigm that provides many concepts, such as **inheritance**, **data binding**, **polymorphism**, etc.
* **Simula** is considered the first object-oriented programming language. The programming paradigm where everything is represented as an object is known as a truly object-orient­­­ed programming language.
* **Smalltalk** is considered the first truly object-oriented programming language.
* The popular object-oriented languages are [Java](https://www.javatpoint.com/java-tutorial), [C#](https://www.javatpoint.com/c-sharp-tutorial), [PHP](https://www.javatpoint.com/php-tutorial), [Python](https://www.javatpoint.com/python-tutorial), [C++](https://www.javatpoint.com/cpp-tutorial), etc.
* **The main aim of object-oriented programming is to implement real-world entities, for example, object, classes, abstraction, inheritance, polymorphism, etc.**

## OOPs (Object-Oriented Programming System)

**Object** means a real-world entity such as a pen, chair, table, computer, watch, etc. **Object-Oriented Programming** **is a methodology or paradigm to design a program using classes and objects.** It simplifies software development and maintenance by providing some concepts:

* [Object](https://www.javatpoint.com/object-and-class-in-java)
* Class
* [Inheritance](https://www.javatpoint.com/inheritance-in-java)
* [Polymorphism](https://www.javatpoint.com/runtime-polymorphism-in-java)
* [Abstraction](https://www.javatpoint.com/abstract-class-in-java)
* [Encapsulation](https://www.javatpoint.com/encapsulation)

Apart from these concepts, there are some other terms which are used in Object-Oriented design:

* Coupling
* Cohesion
* Association
* Aggregation
* Composition



**Object**

* Any entity that has state and behavior is known as an object. For example, a chair, pen, table, keyboard, bike, etc. It can **be physical or logical**.
* **An Object can be defined as an instance of a class. An object contains an address and takes up some space in memory. Objects can communicate without knowing the details of each other's data or code.**

**Example:** A dog is an object because it has states like color, name, breed, etc. as well as behaviors like wagging the tail, barking, eating, etc.

**Class**

* ***Collection of objects* is called class. It is a logical entity.**
* **A class can also be defined as a blueprint from which you can create an individual object. Class doesn't consume any space.**

**Inheritance**

***When one object acquires all the properties and behaviors of a parent object*, it is known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.**

**Lets take eg family where ale property of a parent acquired by the child & child also have there own property as well.**

**Class Animal**

**{**

**Void fun()**

**}**

**Class Cat extends Animal**

**{**

**Void fun()**

**{**

**}**

**}**

**Polymorphism**

* If *one task is performed in different ways*, it is known as polymorphism. For example: to convince the customer differently, to draw something, for example, shape, triangle, rectangle, etc.
* Another example can be to speak something; for example, a cat speaks meow, dog barks woof, etc.
* In Java, we use method overloading and method overriding is use to to achieve polymorphism.

Method overloding—

Class animal

{

Method()

Methd(a,b)

}

Psvm()

{

Animal a = new animal()

a. method()

a. method(a,b)

}

1. **class** Vehicle{
2. //defining a method
3. **void** run(){System.out.println("Vehicle is running");}
4. }
5. //Creating a child class
6. **class** Bike2 **extends** Vehicle{
7. //defining the same method as in the parent class
8. **void** run(){System.out.println("Bike is running safely");}
10. **public** **static** **void** main(String args[]){
11. Bike2 obj = **new** Bike2();//creating object
12. obj.run();//calling method
13. }
14. }

**Abstraction**

* *Hiding internal details and showing functionality* is known as abstraction. For example phone call, ATM Machin we don't know the internal processing.
* In Java, we use abstract class and interface to achieve abstraction.

**Encapsulation**

* *Binding (or wrapping) code and data together into a single unit are known as encapsulation*. For example, a capsule, it is wrapped with different medicines.
* A java class is the example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here.



Coupling

Coupling refers to the knowledge or information or dependency of another class. It arises when classes are aware of each other. If a class has the details information of another class, there is strong coupling. In Java, we use private, protected, and public modifiers to display the visibility level of a class, method, and field. You can use interfaces for the weaker coupling because there is no concrete implementation.

**Cohesion**

Cohesion refers to the level of a component which performs a single well-defined task. A single well-defined task is done by a highly cohesive method. The weakly cohesive method will split the task into separate parts. The java.io package is a highly cohesive package because it has I/O related classes and interface. However, the java.util package is a weakly cohesive package because it has unrelated classes and interfaces.

**Association**

Association represents the relationship between the objects. Here, one object can be associated with one object or many objects. There can be four types of association between the objects:

* One to One
* One to Many
* Many to One, and
* Many to Many

Let's understand the relationship with real-time examples. For example, One country can have one prime minister (one to one), and a prime minister can have many ministers (one to many). Also, many MP's can have one prime minister (many to one), and many ministers can have many departments (many to many).

**Association can be undirectional or bidirectional.**

**Aggregation**

* Aggregation is a way to achieve Association.
* It represents the **weak relationship** between objects.
* It is also termed as a ***has-a* relationship** in Java.
* Like, **inheritance represents the *is-a* relationship**. It is another way to reuse objects.

**Composition**

* The composition is also a way to achieve Association.
* The composition represents the relationship where one object contains other objects as **a part of** its state.
* There is a strong relationship between the containing object and the dependent object.
* It is the state where containing objects do not have an independent existence.
* If you delete the parent object, all the child objects will be deleted automatically.

**Advantage of OOPs over Procedure-oriented programming language**

1) OOPs makes development and maintenance easier, whereas, in a procedure-oriented programming language, it is not easy to manage if code grows as project size increases.

2) OOPs provides data hiding, whereas, in a procedure-oriented programming language, global data can be accessed from anywhere.

|  |  |
| --- | --- |
| Global Data  Figure: Data Representation in Procedure-Oriented Programming | Object Data  Figure: Data Representation in Object-Oriented Programming |

3) OOPs provides the ability to simulate real-world event much more effectively. We can provide the solution of real word problem if we are using the Object-Oriented Programming language.

**What is the difference between an object-oriented programming language and object-based programming language?**

Object-based programming language follows all the features of OOPs except Inheritance. JavaScript and VBScript are examples of object-based programming languages.

# *Java Naming conventions*

===============================================================================

* Java naming convention is a rule to follow as you decide what to name your identifiers such as class, package, variable, constant, method, etc.
* But, it is not forced to follow. So, it is known as convention not rule. These conventions are suggested by several Java communities such as Sun Microsystems and Netscape.
* All the classes, interfaces, packages, methods and fields of Java programming language are given according to the Java naming convention. If you fail to follow these conventions, it may generate confusion or erroneous code.

Advantage of naming conventions in java

By using standard Java naming conventions, you make your code easier to read for yourself and other programmers. Readability of Java program is very important. It indicates that less time is spent to figure out what the code does.

The following are the key rules that must be followed by every identifier:

* The name must not contain any white spaces.
* The name should not start with special characters like & (ampersand), $ (dollar), \_ (underscore).

Let's see some other rules that should be followed by identifiers.

Class

* It should start with the uppercase letter.
* It should be a noun such as Color, Button, System, Thread, etc.
* Use appropriate words, instead of acronyms.

**Example: -**

1. **public** **class** Employee
2. {
3. //code snippet
4. }

Interface

* It should start with the uppercase letter.
* It should be an adjective such as Runnable, Remote, ActionListener.
* Use appropriate words, instead of acronyms.

**Example: -**

1. **interface** Printable
2. {
3. //code snippet
4. }

Method

* It should start with lowercase letter.
* It should be a verb such as main(), print(), println().
* If the name contains multiple words, start it with a lowercase letter followed by an uppercase letter such as actionPerformed().

**Example:-**

1. **class** Employee
2. {
3. //method
4. **void** draw()
5. {
6. //code snippet
7. }
8. }

Variable

* It should start with a lowercase letter such as id, name.
* It should not start with the special characters like & (ampersand), $ (dollar), \_ (underscore).
* If the name contains multiple words, start it with the lowercase letter followed by an uppercase letter such as firstName, lastName.
* Avoid using one-character variables such as x, y, z.

**Example :-**

1. **class** Employee
2. {
3. //variable
4. **int** id;
5. //code snippet
6. }

Package

* It should be a lowercase letter such as java, lang.
* If the name contains multiple words, it should be separated by dots (.) such as java.util, java.lang.

**Example :-**

1. **package** com.javatpoint; //package
2. **class** Employee
3. {
4. //code snippet
5. }

Constant

* It should be in uppercase letters such as RED, YELLOW.
* If the name contains multiple words, it should be separated by an underscore(\_) such as MAX\_PRIORITY.
* It may contain digits but not as the first letter.

**Example :-**

1. **class** Employee
2. {
3. //constant
4. **static** **final** **int** MIN\_AGE = 18;
5. //code snippet
6. }

CamelCase in java naming conventions

* Java follows camel-case syntax for naming the class, interface, method, and variable.
* If the name is combined with two words, the second word will start with uppercase letter always such as actionPerformed(), firstName, ActionEvent, ActionListener, etc.

# *Objects ,Classes and Method in Java*

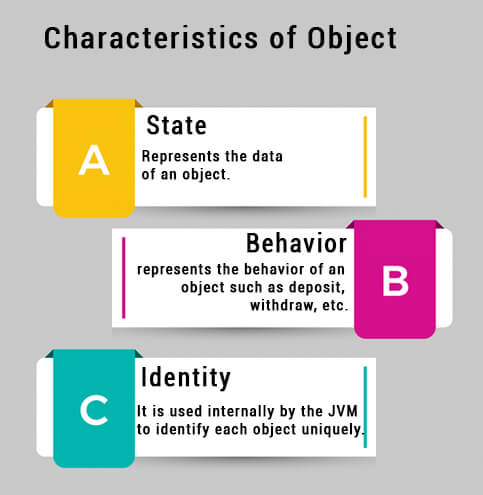
====================================================================================

### *What is an object in Java:*

* **An entity that has state and behavior is known as an object** e.g., chair, bike, marker, pen, table, car, etc. It can be physical or logical (tangible and intangible). The example of an intangible object is the banking system.

An object has three characteristics:

* **State:** represents the data (value) of an object.
* **Behavior:** represents the behavior (functionality) of an object such as deposit, withdraw, etc.
* **Identity:** An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, it is used internally by the JVM to identify each object uniquely.



For Example, Pen is an object. Its name is Reynolds; color is white, known as its state. It is used to write, so writing is its behavior.

Object-pen

State-name(Reynolds),color blue

Behaviour-writing

**An object is an instance of a class.** A class is a template or blueprint from which objects are created. So, an object is the instance(result) of a class.

**Object Definitions:**

* An object is *a real-world entity*.
* An object is *a runtime entity*.
* The object is *an entity which has state and behavior*.
* ***The object is an instance of a class.***

***What is a class in Java:***

* A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical.

### A class is a template or blueprint from which objects are created. So, an object is the instance(result) of a class.

A class in Java can contain:

|  |  |
| --- | --- |
| * **Fields** * **Methods** * **Constructors** * **Blocks** * **Nested class and interface** | Class in Java |

**Syntax to declare a class:**

1. **class <class\_name>{**
2. **field;**
3. **method;**
4. **}**

### Instance variable in Java

* + A variable which is **created inside the class but outside the method** is known as an instance variable.
  + Instance variable doesn't get memory at compile time.
  + It gets memory at runtime when an object or instance is created.
  + That is why it is known as an instance variable.

### *Method in Java:*

In Java, a method is like a function which is used to expose the behavior of an object.

#### Advantage of Method

* Code Reusability
* Code Optimization

### *new keyword in Java:*

The new keyword is used to allocate memory at runtime. All objects get memory in **Heap** memory area.

* **Object and Class Example: main within the class**
* In this example, we have created a Student class which has two data members id and name. We are creating the object of the Student class by new keyword and printing the object's value.
* Here, we are creating a main() method inside the class.

*File: Student.java*

1. //Java Program to illustrate how to define a class and fields
2. //Defining a Student class.
3. **class** Student{
4. //defining fields
5. **int** id;//field or data member or instance variable
6. String name;
7. //creating main method inside the Student class
8. **public** **static** **void** main(String args[]){
9. //Creating an object or instance
10. Student s1=**new** Student();//creating an object of Student
11. //Printing values of the object
12. System.out.println(s1.id);//accessing member through reference variable
13. System.out.println(s1.name);
14. }
15. }

Output:

0

null

***Object and Class Example: main outside the class***

* In real time development, we create classes and use it from another class. It is a better approach than previous one. Let's see a simple example, where we are having main() method in another class.
* We can have multiple classes in different Java files or single Java file. If you define multiple classes in a single Java source file, it is a good idea to save the file name with the class name which has main() method.

*File: TestStudent1.java*

1. //Java Program to demonstrate having the main method in
2. //another class
3. //Creating Student class.
4. **class** Student{
5. **int** id;
6. String name;
7. }
8. //Creating another class TestStudent1 which contains the main method
9. **class** TestStudent1{
10. **public** **static** **void** main(String args[]){
11. Student s1=**new** Student();
12. System.out.println(s1.id);
13. System.out.println(s1.name);
14. }
15. }

Output:

0

null

***3 Ways to initialize object***

***There are 3 ways to initialize object in Java.***

1. By reference variable
2. By method
3. By constructor

* ***Object and Class Example: Initialization through reference***

Initializing an object means storing data into the object. Let's see a simple example where we are going to initialize the object through a reference variable.

*File: TestStudent2.java*

1. **class** Student{
2. **int** id;
3. String name;
4. }
5. **class** TestStudent2{
6. **public** **static** **void** main(String args[]){
7. Student s1=**new** Student();
8. s1.id=101;
9. s1.name="Sonoo";
10. System.out.println(s1.id+" "+s1.name);//printing members with a white space
11. }
12. }

Output:

101 Sonoo

***We can also create multiple objects and store information in it through reference variable***.

*File: TestStudent3.java*

1. **class** Student{
2. **int** id;
3. String name;
4. }
5. **class** TestStudent3{
6. **public** **static** **void** main(String args[]){
7. //Creating objects
8. Student s1=**new** Student();
9. Student s2=**new** Student();
10. //Initializing objects
11. s1.id=101;
12. s1.name="Sonoo";
13. s2.id=102;
14. s2.name="Amit";
15. //Printing data
16. System.out.println(s1.id+" "+s1.name);
17. System.out.println(s2.id+" "+s2.name);
18. }
19. }

***Object and Class Example: Initialization through method***

* In this example, we are creating the two objects of Student class and initializing the value to these objects by invoking the insertRecord method. Here, we are displaying the state (data) of the objects by invoking the displayInformation() method.

*File: TestStudent4.java*

1. **class** Student{
2. **int** rollno;
3. String name;
4. **void** insertRecord(**int** r, String n){
5. rollno=r;
6. name=n;
7. }
8. **void** displayInformation(){System.out.println(rollno+" "+name);}
9. }
10. **class** TestStudent4{
11. **public** **static** **void** main(String args[]){
12. Student s1=**new** Student();
13. Student s2=**new** Student();
14. s1.insertRecord(111,"Karan");
15. s2.insertRecord(222,"Aryan");
16. s1.displayInformation();
17. s2.displayInformation();
18. }
19. }



* As you can see in the above figure, object gets the memory in heap memory area. The reference variable refers to the object allocated in the heap memory area. Here, s1 and s2 both are reference variables that refer to the objects allocated in memory.

***Object and Class Example: Initialization through a constructor***

Object and Class Example: Employee

Let's see an example where we are maintaining records of employees.

*File: TestEmployee.java*

1. **class** Employee{
2. **int** id;
3. String name;
4. **float** salary;
5. void Employee (**int** i, String n, **float** s) {
6. id=i;
7. name=n;
8. salary=s;
9. }
10. **void** display(){System.out.println(id+" "+name+" "+salary);}
11. }
12. **public** **class** TestEmployee {
13. **public** **static** **void** main(String[] args) {
14. Employee e1=**new** Employee();
15. Employee e2=**new** Employee();
16. Employee e3=**new** Employee();
17. e1.insert(101,"ajeet",45000);
18. e2.insert(102,"irfan",25000);
19. e3.insert(103,"nakul",55000);
20. e1.display();
21. e2.display();
22. e3.display();
23. }
24. }

***Object and Class Example: Rectangle***

There is given another example that maintains the records of Rectangle class.

*File: TestRectangle1.java*

1. **class** Rectangle{
2. **int** length;
3. **int** width;
4. **void** insert(**int** l, **int** w){
5. length=l;
6. width=w;
7. }
8. **void** calculateArea(){System.out.println(length\*width);}
9. }
10. **class** TestRectangle1{
11. **public** **static** **void** main(String args[]){
12. Rectangle r1=**new** Rectangle();
13. Rectangle r2=**new** Rectangle();
14. r1.insert(11,5);
15. r2.insert(3,15);
16. r1.calculateArea();
17. r2.calculateArea();
18. }
19. }

**What are the different ways to create an object in Java?**

There are many ways to create an object in java. They are:

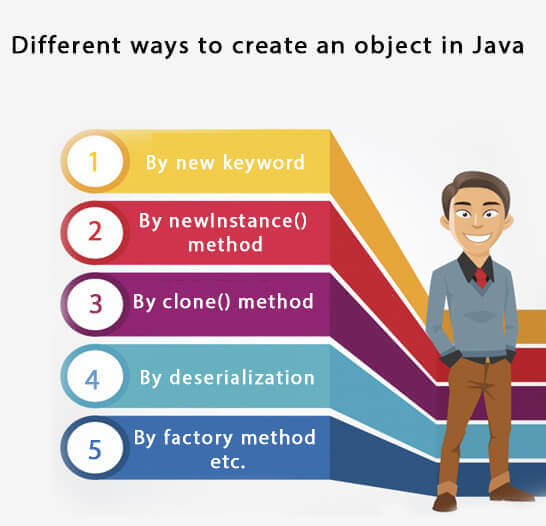
* By new keyword
* By new Instance() method
* By clone() method
* By deserialization
* By factory method etc.

**Anonymous object**

* Anonymous simply means nameless. An object which has **no reference** is known as an anonymous object. It can be used at the time of object creation only.
* **If you have to use an object only once, an anonymous object is a good approach.**

For example:

**new** Calculation();//anonymous object



# *Constructors in Java*

=============================================================================

* **It is a function of the class.**
* **Its name is similar to the class name.**
* **There is no return type for constructor.**
* ***Used to initialize the data of the object at the time of creation of object.***
* **A Java constructor cannot be abstract, static, final, and synchronized**
* In [Java](https://www.javatpoint.com/java-tutorial), a constructor is a block of codes similar to the method. It is called when an instance of the [class](https://www.javatpoint.com/object-and-class-in-java) is created. At the time of calling constructor, memory for the object is allocated in the memory.
* 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.
* There **are two types of constructors** in Java:

**1)no-arg constructor**

**2) parameterized constructor.**

**Note: It is called constructor because it constructs the values at the time of object creation. It is not necessary to write a constructor for a class. It is because java compiler creates a default constructor if your class doesn't have any.**

Rules for creating Java constructor

There are two rules defined for the constructor.

1. **Constructor name must be the same as its class name**
2. **A Constructor must have no explicit return type**
3. **A Java constructor cannot be abstract, static, final, and synchronized**

Note: We can use [access modifiers](https://www.javatpoint.com/access-modifiers) while declaring a constructor. It controls the object creation. In other words, we can have private, protected, public or default constructor in Java.

Types of Java constructors

There are two types of constructors in Java:

1. Default constructor (no-arg constructor)
2. Parameterized constructor



Java Default Constructor

A constructor is called "Default Constructor" when it doesn't have any parameter.

Syntax of default constructor:

1. <class\_name>(){}

Example of default constructor

|  |
| --- |
| In this example, we are creating the no-arg constructor in the Bike class. It will be invoked at the time of object creation. |

1. //Java Program to create and call a default constructor
2. **class** Bike1{
3. //creating a default constructor
4. Bike1()
5. {
6. System.out.println("Bike is created");
7. }
8. //main method
9. **public** **static** **void** main(String args[]){
10. //calling a default constructor
11. Bike1 b=**new** Bike1();
12. }
13. }

Output:

Bike is created

**Rule:** If there is no constructor in a class, compiler automatically creates a default constructor.



Q) What is the purpose of a default constructor?

The default constructor is used to provide the default values to the object like 0, null, etc., depending on the type.

Example of default constructor that displays the default values

1. //Let us see another example of default constructor
2. //which displays the default values
3. **class** Student3{
4. **int** id;
5. String name;
6. //method to display the value of id and name
7. **void** display(){System.out.println(id+" "+name);}
9. **public** **static** **void** main(String args[]){
10. //creating objects
11. Student3 s1=**new** Student3();
12. Student3 s2=**new** Student3();
13. //displaying values of the object
14. s1.display();
15. s2.display();
16. }
17. }

Output:

0 null

0 null

**Explanation:**

In the above class,you are not creating any constructor so compiler provides you a default constructor. Here 0 and null values are provided by default constructor.

Java Parameterized Constructor

* A constructor which has a specific number of parameters is called a parameterized constructor.

Why use the parameterized constructor?

* The parameterized constructor is used to provide different values to distinct objects. However, you can provide the same values also.

Example of parameterized constructor

* In this example, we have created the constructor of Student class that have two parameters. We can have any number of parameters in the constructor.

1. //Java Program to demonstrate the use of the parameterized constructor.
2. **class** Student4{
3. **int** id;
4. String name;
5. //creating a parameterized constructor
6. Student4(**int** i,String n){
7. id = i;
8. name = n;
9. }
10. //method to display the values
11. **void** display(){System.out.println(id+" "+name);}
13. **public** **static** **void** main(String args[]){
14. //creating objects and passing values
15. Student4 s1 = **new** Student4(111,"Karan");
16. Student4 s2 = **new** Student4(222,"Aryan");
17. //calling method to display the values of object
18. s1.display();
19. s2.display();
20. }
21. }

Output:

111 Karan

222 Aryan

**Constructor Overloading in Java**

In Java, a constructor is just like a method but without return type. It can also be overloaded like Java methods.

Constructor [overloading in Java](https://www.javatpoint.com/method-overloading-in-java) is a technique of having more than one constructor with different parameter lists. They are arranged in a way that each constructor performs a different task. They are differentiated by the compiler by the number of parameters in the list and their types.

Example of Constructor Overloading

1. //Java program to overload constructors
2. **class** Student5{
3. **int** id;
4. String name;
5. **int** age;
6. //creating two arg constructor
7. Student5(**int** i,String n){
8. id = i;
9. name = n;
10. }
11. //creating three arg constructor
12. Student5(**int** i,String n,**int** a){
13. id = i;
14. name = n;
15. age=a;
16. }
17. **void** display(){System.out.println(id+" "+name+" "+age);}
19. **public** **static** **void** main(String args[]){
20. Student5 s1 = **new** Student5(111,"Karan");
21. Student5 s2 = **new** Student5(222,"Aryan",25);
22. s1.display();
23. s2.display();
24. }
25. }

Output:

111 Karan 0

222 Aryan 25

## Difference between constructor and method in Java

There are many differences between constructors and methods. They are given below.

|  |  |
| --- | --- |
| **Java Constructor** | **Java Method** |
| A constructor is used to initialize the **state** of an object. | A method is used to expose the **behavior** of an object. |
| A constructor must not have a return type. | A method must have a return type. |
| The constructor is invoked implicitly(call By its own). | The method is invoked explicitly(we need to call). |
| The Java compiler provides a default constructor if you don't have any constructor in a class. | The method is not provided by the compiler in any case. |
| The constructor name must be same as the class name. | The method name may or may not be same as the class name. |

Constructor Channing::



**Java Copy Constructor**

* There is no copy constructor in Java. However, we can copy the values from one object to another like copy constructor in C++.
* There are many ways to copy the values of one object into another in Java. They are:
* By constructor
* By assigning the values of one object into another
* By clone() method of Object class

In this example, we are going to copy the values of one object into another using Java constructor.

1. //Java program to initialize the values from one object to another object.
2. **class** Student6{
3. **int** id;
4. String name;
5. //constructor to initialize integer and string
6. Student6(**int** i,String n){
7. id = i;
8. name = n;
9. }
10. //constructor to initialize another object
11. **Student6(Student6 s**){
12. **id = s.id;**
13. name =s.name;
14. }
15. **void** display(){System.out.println(id+" "+name);}
17. **public** **static** **void** main(String args[]){
18. Student6 s1 = **new** Student6(111,"Karan");
19. **Student6 s2 = new Student6(s1);**
20. s1.display();
21. s2.display();
22. }
23. }

Output:

111 Karan

111 Karan

**Copying values without constructor**

We can copy the values of one object into another by assigning the objects values to another object. In this case, there is no need to create the constructor.

1. **class** Student7{
2. **int** id;
3. String name;
4. Student7(**int** i,String n){
5. id = i;
6. name = n;
7. }
8. Student7(){}
9. **void** display(){System.out.println(id+" "+name);}
11. **public** **static** **void** main(String args[]){
12. Student7 s1 = **new** Student7(111,"Karan");
13. Student7 s2 = **new** Student7();
14. **s2.id=s1.id;**  //**copying the1 object value to other without using constructor**
15. s2.name=s1.name;
16. s1.display();
17. s2.display();
18. }
19. }

Output:

111 Karan

111 Karan

Q) Does constructor return any value?

Yes, it is the current class instance (You cannot use return type yet it returns a value).

Can constructor perform other tasks instead of initialization?

Yes, like object creation, starting a thread, calling a method, etc. You can perform any operation in the constructor as you perform in the method.

Is there Constructor class in Java?

Yes.

What is the purpose of Constructor class?

Java provides a Constructor class which can be used to get the internal information of a constructor in the class. It is found in the java.lang.reflect package.

# *Java static keyword*

=============================================================================

* The **static keyword** in [Java](https://www.javatpoint.com/java-tutorial) is used for memory management mainly. We can apply static keyword with [variables](https://www.javatpoint.com/java-variables), methods, blocks and [nested classes](https://www.javatpoint.com/java-inner-class). The static keyword belongs to the class than an instance of the class.

The static can be:

1. Variable (also known as a class variable)
2. Method (also known as a class method)
3. Block
4. Nested class

## 1) Java static variable

* If you declare any variable as static, it is known as a static variable.
* The static variable can be used to refer to the common property of all objects (which is not unique for each object), for example, the company name of employees, college name of students, etc.
* **The static variable gets memory only once in the class area at the time of class loading.**



Advantages of static variable

It makes your program **memory efficient** (i.e., it saves memory).

Understanding the problem without static variable

1. **class** Student{
2. **int** rollno;
3. String name;
4. String college="ITS";
5. }

Suppose there are 500 students in my college, now all instance data members will get memory each time when the object is created. All students have its unique rollno and name, so instance data member is good in such case. Here, "college" refers to the common property of all [objects](https://www.javatpoint.com/object-and-class-in-java). If we make it static, this field will get the memory only once.

**Java static property is shared to all objects.**

Example of static variable

1. //Java Program to demonstrate the use of static variable
2. **class** Student{
3. **int** rollno;//instance variable
4. String name;
5. **static** String college ="ITS";//static variable
6. //constructor
7. Student(**int** r, String n){
8. rollno = r;
9. name = n;
10. }
11. //method to display the values
12. **void** display (){System.out.println(rollno+" "+name+" "+college);}
13. }
14. //Test class to show the values of objects
15. **public** **class** TestStaticVariable1{
16. **public** **static** **void** main(String args[]){
17. Student s1 = **new** Student(111,"Karan");
18. Student s2 = **new** Student(222,"Aryan");
19. //we can change the college of all objects by the single line of code
20. //Student.college="BBDIT";
21. s1.display();
22. s2.display();
23. }
24. }

Output:

111 Karan ITS

222 Aryan ITS



### Program of the counter without static variable

In this example, we have created an instance variable named count which is incremented in the constructor. Since instance variable gets the memory at the time of object creation, each object will have the copy of the instance variable. If it is incremented, it won't reflect other objects. So each object will have the value 1 in the count variable.

1. //Java Program to demonstrate the use of an instance variable
2. //which get memory each time when we create an object of the class.
3. **class** Counter{
4. **int** count=0;//will get memory each time when the instance is created
6. Counter(){
7. count++;//incrementing value
8. System.out.println(count);
9. }
11. **public** **static** **void** main(String args[]){
12. //Creating objects
13. Counter c1=**new** Counter();
14. Counter c2=**new** Counter();
15. Counter c3=**new** Counter();
16. }
17. }

Output:

1

1

1

**Program of counter by static variable**

As we have mentioned above, static variable will get the memory only once, if any object changes the value of the static variable, it will retain its value.

1. //Java Program to illustrate the use of static variable which
2. //is shared with all objects.
3. **class** Counter2{
4. **static** **int** count=0;//will get memory only once and retain its value
6. Counter2(){
7. count++;//incrementing the value of static variable
8. System.out.println(count);
9. }
11. **public** **static** **void** main(String args[]){
12. //creating objects
13. Counter2 c1=**new** Counter2();
14. Counter2 c2=**new** Counter2();
15. Counter2 c3=**new** Counter2();
16. }
17. }

Output:

1

2

3

**2) Java static method**

If you apply static keyword with any method, it is known as static method.

* A static method belongs to the class rather than the object of a class.
* A static method can be invoked without the need for creating an instance of a class.
* A static method can access static data member and can change the value of it.

Example of static method

1. //Java Program to demonstrate the use of a static method.
2. **class** Student{
3. **int** rollno;
4. String name;
5. **static** String college = "ITS";
6. //static method to change the value of static variable
7. **static** **void** change(){
8. college = "BBDIT";
9. }
10. //constructor to initialize the variable
11. Student(**int** r, String n){
12. rollno = r;
13. name = n;
14. }
15. //method to display values
16. **void** display(){System.out.println(rollno+" "+name+" "+college);}
17. }
18. //Test class to create and display the values of object
19. **public** **class** TestStaticMethod{
20. **public** **static** **void** main(String args[]){
21. **Student.change();//**calling change method
22. //creating objects
23. Student s1 = **new** Student(111,"Karan");
24. Student s2 = **new** Student(222,"Aryan");
25. Student s3 = **new** Student(333,"Sonoo");
26. //calling display method
27. s1.display();
28. s2.display();
29. s3.display();
30. }
31. }

Output:111 Karan BBDIT

222 Aryan BBDIT

333 Sonoo BBDIT

Another example of a static method that performs a normal calculation

1. //Java Program to get the cube of a given number using the static method
3. **class** Calculate{
4. **static** **int** cube(**int** x){
5. **return** x\*x\*x;
6. }
8. **public** **static** **void** main(String args[]){
9. **int result=Calculate.cube(5);**
10. System.out.println(result);
11. }
12. }

Output:125

**Restrictions for the static method**

There are two main restrictions for the static method. They are:

* **The static method can not use non static data member or call non-static method directly.**
* **this and super cannot be used in static context.**

1. **class** A{
2. **int** a=40;//non static
4. **public** **static** **void** main(String args[]){
5. System.out.println(a);
6. }
7. }

Output:Compile Time Error

Q) Why is the Java main method static?

It is because the object is not required to call a static method. If it were a non-static method, [JVM](https://www.javatpoint.com/jvm-java-virtual-machine) creates an object first then call main() method that will lead the problem of extra memory allocation.

**3) Java static block**

* Is used to initialize the static data member.
* It is executed before the main method at the time of class loading.

Example of static block

1. **class** A2{
2. **static**{System.out.println("static block is invoked");}
3. **public** **static** **void** main(String args[]){
4. System.out.println("Hello main");
5. }
6. }

Output:

static block is invoked

Hello main

Q) Can we execute a program without main() method?

No, one of the ways was the static block, but it was possible till JDK 1.6. Since JDK 1.7, it is not possible to execute a Java class without the [main method](https://www.javatpoint.com/java-main-method).

1. **class** A3{
2. **static**{
3. System.out.println("static block is invoked");
4. System.exit(0);
5. }
6. }

Output:

static block is invoked

Since JDK 1.7 and above, output would be:

Error: Main method not found in class A3, please define the main method as:

public static void main(String[] args)

or a JavaFX application class must extend javafx.application.Application

|  |
| --- |
| ====================================Shiv nath sir=============================  /\*\*to share common data we define static  filename: Staticmethod.java  //To access static variable we don’t need to create any objects.  //non-static members - means related to obj so we need object for that  //static members - means related to entire class so we don’t need obj for that. We need class only.  //static block is used to initialize the static data & members  //static block is executed only once whenever class is first time used(loaded)  //we can access any static part of the class from the non-static context (Block& member)but reverse is not true. Meaning we cant access any non-static part of the class from the static context  (non-static data & member)  \* //It is executed in once in a life  \* //Static block execute 1 st then non static will execute their order does not mattter always static will execute first  //After static block constructor will execute then non static block will execute fi question of this which pint will be first or printing order  \* class Test  {  static int a; //this is not a part of object it is commomn for entire class.  int b; //non-static/instance variable  int c; //non-static/instance variable    {  //will be called each time when obje is creted.  //non-static block  sop("non-static block");  }  static  {  //used to inialized or give value to static data.  //static block:used to initialize or to give value to the static data. //cannot used in method.    sop("static block");  a=20;    }  void fun()  {  sop(a);  sop("Fun of test");      }  //Static function (class method) is mainly used to work on static data.  //it not allowed non static variable to access eg b,c.  static void anotherFun()  {  sop(a);  sop(b); //error  sop(c); //error  fun(); //error  sop("anotherFun of test");  }  }  class StaticDemo  {  p.s.v.m(String args[])  {  //new Test();  Test.anotherFun();  //new Test();  s.o.p(Test.a); //print 0  Test t = new Test();  Test.fun();  Test.b;  Test.c;  t.fun(); //works  sop(t.b); //works  }  }  //static member (data and member)can using classname or object.  //any non-static can access static data  can any static method acces non-static data ?  no a static method can access/use only static data & membes  Test() require 8 byte space in memory for object (a,b are object only).  u can declare function and data as a static u will simply use that data by using class  to access non-static data we need objects.  static we can acess using class name  method can be static ans non-static  we declare method is static bcz -in that to access we dont need any object \*\*/  /\*\*Filename: StaticMethod.java  //To access the static variable we don't need to create any object.  // Non-static member - means related to object so we need object for that.  // Static member - means related to entire class so we do not need object for that. We need class only.  class Test  {  static int a ; //It is also known as class variable/data  int b; //non-static or instance variable  int c; //non-static or instance variable  {  //non-static block : will be called each and every time when any object is created.  System.out.println("Within non-static block");  b = 5;  c = 10;  }  static  {  //static block : Used to initialize or give value to the static data.  //This block is executed only once at the time of loading of class.  System.out.println("Within static block");  a = 20;  }  Test()  {  System.out.println("No argument constructor");  }  void fun()  {  s.o.p(a); //Works  System.out.println("fun of Test");  }  //static function (class method) is mainly used to work upon static data.  static void anotherFun()  {  s.o.p(a);  s.o.p(b); // Error  s.o.p(c); // Error  fun(); // Error  System.out.println("anotherFun of Test");  }  }  class StaticDemo  {  public static void main(String args[])  {  //new Test();  Test.anotherFun();  //new Test();    }  }  1. Static member (data and functions) can be accessed using class name or object.  2. Any non-static method can access static data.  3. Can any static method access the non-static (instance variable) data? NO  4. A static method can access/use only static data and methods.  5. Static block is used to initialize the static data members.  6. Static block is executed only once whenever class is first time used (loaded).  7. We can access any static part of the class from the non-static context (static block or method) but reverse is not true. Meaning we can not access any non-static part of the class from the static context (non-static block or method)  \*\*/  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  /\*Static Overriden is not possible  ***Static Method in parent and Child :::***  class First  {  static void fun()  {    }  }  class Second extends First  {  static void fun()  {    }    }  Class Demo  {  public static void main(String args[])  {  First f = new Second();  f.fun(); //Which method will be called -->first() of fun is called    }  }  Static it will depend of the rteference type i.e First F it will call first fun ()functoiom  Static has always compile time bindinh  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  In Non static sunction it will depend upon the new second i.e reference stored and it will call second fun() dunction  class First  {  void fun()  {    }  }  class Second extends First  {  void fun()  {    }    }  Class Demo  {  public static void main(String args[])  {  First f = new Second();  f.fun(); //Which method will be called -->Second() fun is called  }  } |

# *this keyword in java*

**variable shadowing-===--**

1.If the instance variable and local variable have same name whenever you print (access) it in the method. The value of the local variable will be printed (shadowing the instance variable).

2.local is hide by instance variable.

3.When local & instance have same name then local variable hides the instance variable or we can say instance variable is being shadowing by local variable.

# *this keyword in java:*

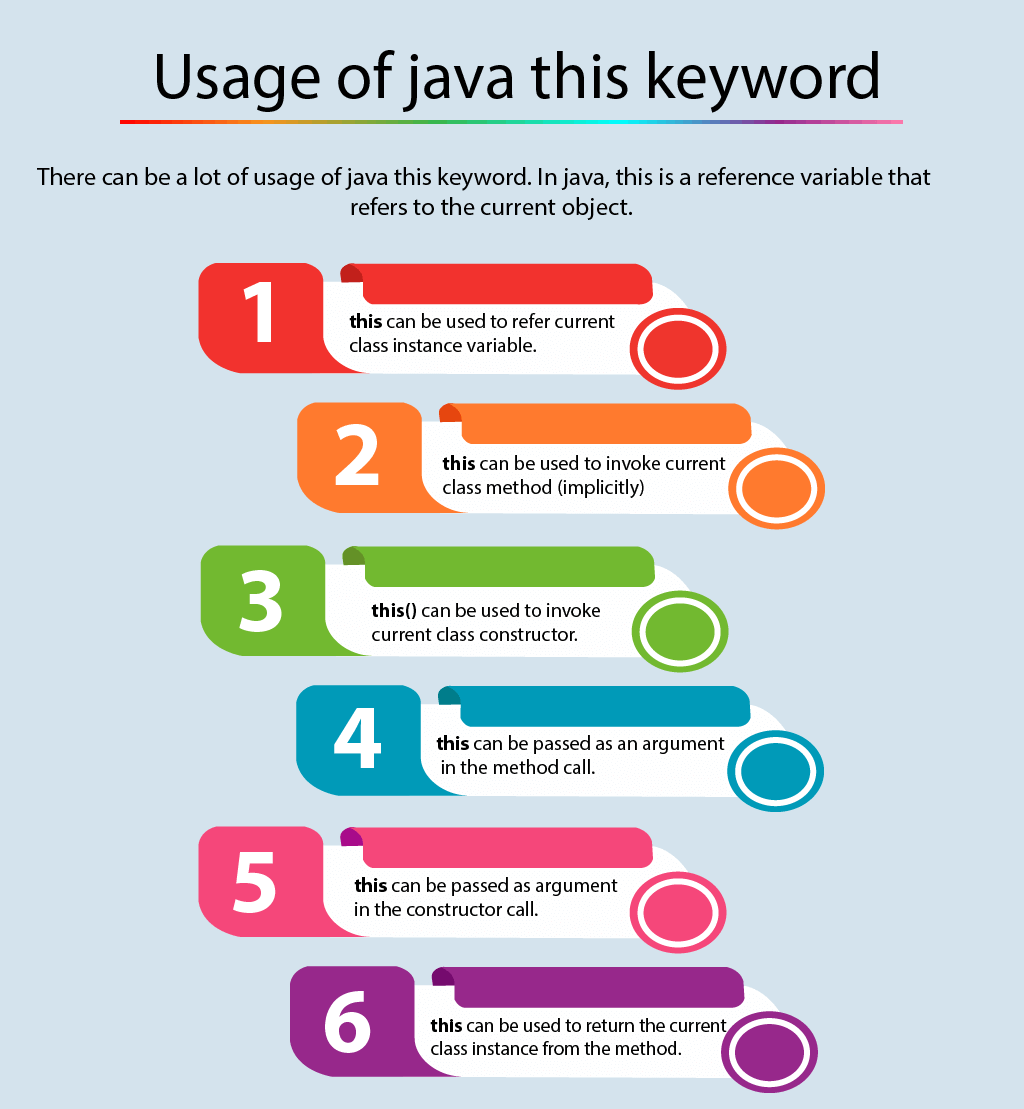
There can be a lot of usage of **java this keyword**. In java, this is a **reference variable** that refers to the current object.



## Usage of java this keyword

Here is given the 6 usage of java this keyword.

1. this can be used to refer current class instance variable.
2. this can be used to invoke current class method (implicitly)
3. this() can be used to invoke current class constructor.
4. this can be passed as an argument in the method call.
5. this can be passed as argument in the constructor call.
6. this can be used to return the current class instance from the method.



1) this: to refer current class instance variable

The this keyword can be used to refer current class instance variable. If there is ambiguity between the instance variables and parameters, this keyword resolves the problem of ambiguity.

Understanding the problem without this keyword

|  |
| --- |
| Let's understand the problem if we don't use this keyword by the example given below: |

1. **class** Student{
2. **int** rollno;
3. String name;
4. **float** fee;
5. Student(**int** rollno,String name,**float** fee){
6. rollno=rollno;
7. name=name;
8. fee=fee;
9. }
10. **void** display(){System.out.println(rollno+" "+name+" "+fee);}
11. }
12. **class** TestThis1{
13. **public** **static** **void** main(String args[]){
14. Student s1=**new** Student(111,"ankit",5000f);
15. Student s2=**new** Student(112,"sumit",6000f);
16. s1.display();
17. s2.display();
18. }}

Output:

0 null 0.0

0 null 0.0

In the above example, parameters (formal arguments) and instance variables are same. So, we are using this keyword to distinguish local variable and instance variable.

Solution of the above problem by this keyword

1. **class** Student{
2. **int** rollno;
3. String name;
4. **float** fee;
5. Student(**int** rollno,String name,**float** fee){
6. **this**.rollno=rollno;
7. **this**.name=name;
8. **this**.fee=fee;
9. }
10. **void** display(){System.out.println(rollno+" "+name+" "+fee);}
11. }
13. **class** TestThis2{
14. **public** **static** **void** main(String args[]){
15. Student s1=**new** Student(111,"ankit",5000f);
16. Student s2=**new** Student(112,"sumit",6000f);
17. s1.display();
18. s2.display();
19. }}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestThis2)

Output:

111 ankit 5000

112 sumit 6000

If local variables(formal arguments) and instance variables are different, there is no need to use this keyword like in the following program:

Program where this keyword is not required

1. **class** Student{
2. **int** rollno;
3. String name;
4. **float** fee;
5. Student(**int** r,String n,**float** f){
6. rollno=r;
7. name=n;
8. fee=f;
9. }
10. **void** display(){System.out.println(rollno+" "+name+" "+fee);}
11. }
13. **class** TestThis3{
14. **public** **static** **void** main(String args[]){
15. Student s1=**new** Student(111,"ankit",5000f);
16. Student s2=**new** Student(112,"sumit",6000f);
17. s1.display();
18. s2.display();
19. }}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestThis3)

Output:

111 ankit 5000

112 sumit 6000

It is better approach to use meaningful names for variables. So we use same name for instance variables and parameters in real time, and always use this keyword.

2) this: to invoke current class method

You may invoke the method of the current class by using the this keyword. If you don't use the this keyword, compiler automatically adds this keyword while invoking the method. Let's see the example



1. **class** A{
2. **void** m(){System.out.println("hello m");}
3. **void** n(){
4. System.out.println("hello n");
5. //m();//same as this.m()
6. **this**.m();
7. }
8. }
9. **class** TestThis4{
10. **public** **static** **void** main(String args[]){
11. A a=**new** A();
12. a.n();
13. }}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestThis4)

Output:

hello n

hello m

3) this() : to invoke current class constructor

The this() constructor call can be used to invoke the current class constructor. It is used to reuse the constructor. In other words, it is used for constructor chaining.

**Calling default constructor from parameterized constructor:**

1. **class** A{
2. A(){System.out.println("hello a");}
3. A(**int** x){
4. **this**();
5. System.out.println(x);
6. }
7. }
8. **class** TestThis5{
9. **public** **static** **void** main(String args[]){
10. A a=**new** A(10);
11. }}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestThis5)

Output:

hello a

10

**Calling parameterized constructor from default constructor:**

1. **class** A{
2. A(){
3. **this**(5);
4. System.out.println("hello a");
5. }
6. A(**int** x){
7. System.out.println(x);
8. }
9. }
10. **class** TestThis6{
11. **public** **static** **void** main(String args[]){
12. A a=**new** A();
13. }}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestThis6)

Output:

5

hello a

Real usage of this() constructor call

The this() constructor call should be used to reuse the constructor from the constructor. It maintains the chain between the constructors i.e. it is used for constructor chaining. Let's see the example given below that displays the actual use of this keyword.

1. **class** Student{
2. **int** rollno;
3. String name,course;
4. **float** fee;
5. Student(**int** rollno,String name,String course){
6. **this**.rollno=rollno;
7. **this**.name=name;
8. **this**.course=course;
9. }
10. Student(**int** rollno,String name,String course,**float** fee){
11. **this**(rollno,name,course);//reusing constructor
12. **this**.fee=fee;
13. }
14. **void** display(){System.out.println(rollno+" "+name+" "+course+" "+fee);}
15. }
16. **class** TestThis7{
17. **public** **static** **void** main(String args[]){
18. Student s1=**new** Student(111,"ankit","java");
19. Student s2=**new** Student(112,"sumit","java",6000f);
20. s1.display();
21. s2.display();
22. }}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestThis7)

Output:

111 ankit java null

112 sumit java 6000

Rule: Call to this() must be the first statement in constructor.

1. **class** Student{
2. **int** rollno;
3. String name,course;
4. **float** fee;
5. Student(**int** rollno,String name,String course){
6. **this**.rollno=rollno;
7. **this**.name=name;
8. **this**.course=course;
9. }
10. Student(**int** rollno,String name,String course,**float** fee){
11. **this**.fee=fee;
12. **this**(rollno,name,course);//C.T.Error
13. }
14. **void** display(){System.out.println(rollno+" "+name+" "+course+" "+fee);}
15. }
16. **class** TestThis8{
17. **public** **static** **void** main(String args[]){
18. Student s1=**new** Student(111,"ankit","java");
19. Student s2=**new** Student(112,"sumit","java",6000f);
20. s1.display();
21. s2.display();
22. }}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestThis8)

Compile Time Error: Call to this must be first statement in constructor

4) this: to pass as an argument in the method

The this keyword can also be passed as an argument in the method. It is mainly used in the event handling. Let's see the example:

1. **class** S2{
2. **void** m(S2 obj){
3. System.out.println("method is invoked");
4. }
5. **void** p(){
6. m(**this**);
7. }
8. **public** **static** **void** main(String args[]){
9. S2 s1 = **new** S2();
10. s1.p();
11. }
12. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=S2)

Output:

method is invoked

Application of this that can be passed as an argument:

In event handling (or) in a situation where we have to provide reference of a class to another one. It is used to reuse one object in many methods.

5) this: to pass as argument in the constructor call

We can pass the this keyword in the constructor also. It is useful if we have to use one object in multiple classes. Let's see the example:

1. **class** B{
2. A4 obj;
3. B(A4 obj){
4. **this**.obj=obj;
5. }
6. **void** display(){
7. System.out.println(obj.data);//using data member of A4 class
8. }
9. }
11. **class** A4{
12. **int** data=10;
13. A4(){
14. B b=**new** B(**this**);
15. b.display();
16. }
17. **public** **static** **void** main(String args[]){
18. A4 a=**new** A4();
19. }
20. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=A4)

Output:10

6) this keyword can be used to return current class instance

We can return this keyword as an statement from the method. In such case, return type of the method must be the class type (non-primitive). Let's see the example:

Syntax of this that can be returned as a statement

1. return\_type method\_name(){
2. **return** **this**;
3. }

Example of this keyword that you return as a statement from the method

1. **class** A{
2. A getA(){
3. **return** **this**;
4. }
5. **void** msg(){System.out.println("Hello java");}
6. }
7. **class** Test1{
8. **public** **static** **void** main(String args[]){
9. **new** A().getA().msg();
10. }
11. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Test1)

Output:

Hello java

Proving this keyword

|  |
| --- |
| Let's prove that this keyword refers to the current class instance variable. In this program, we are printing the reference variable and this, output of both variables are same. |

1. **class** A5{
2. **void** m(){
3. System.out.println(**this**);//prints same reference ID
4. }
5. **public** **static** **void** main(String args[]){
6. A5 obj=**new** A5();
7. System.out.println(obj);//prints the reference ID
8. obj.m();
9. }
10. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=A5)

Output:

A5@22b3ea59

A5@22b3ea59

|  |
| --- |
| =============================Shiv nath sir====================================  // This keyword - this keyword alwyas keeps the reference of curent object  // This keyword is never called in the Main Class  // When same name given to variable this keyword is used to differentiate  //It is also used to cal from one constructor to another constructor i.e it is clled as chain constructot  //the constructor calling from one to other constructor it may be one arguement ,two or no arguement constructor  //Limitation we can constructor using this keyword if we use this to call the constructor then that may be the very  //first statement within constructor otherwise compiler will gve an error  //U cannt call this in function it is used in constructor to call other constructor  // Variable Shadowing : when local and instance have same name then local variables hides the instance  // variable or we can say that  package Topic;  class This  {    int rollNo ;  String name ;  int age;  double marks;  static String institutename = "CDAC";  This()  {    System.out.println("Zero- Arguement Constructor is Called");  }  This(int rollNo,String name,int age,double marks)  {  this(); //This is used for calling Zero Arguement Constructor  System.out.println("Four- Arguement Constructor is Called");  this.rollNo = rollNo; //In this local variable is differentiated with Instance Variable  this.name = name ;  this.age = age;  this.marks = marks;  }  void learning()  {    System.out.println(name + "is Learning");  }  void writing()  {  System.out.println(name + "is Writing");  }  void givingExam()  {  System.out.println(name + "is giving Exam");  }  void printDetails()  {  System.out.println("Roll :" + rollNo);  System.out.println("Name :" + name);  System.out.println("Age :" + age);  System.out.println("Marks :" + marks);  }  }  class Studen  {  //cretaing main mehod inside the student class  public static void main(String args[])  {  //Creating an object or instance    This s1= new This(87,"Ramesh",25,87); //creating an object of student  s1.printDetails();  System.out.println(s1.institutename);  System.out.println(s1.name);  This s2 ;  s2= new This(); //creating an object of student  s2.printDetails(); //. dot is called Member Access Operator  }  }  /\*\*Use of "this" keyword  =======================  'this' always keeps the reference of the current object.  Q. When we say current object what does it mean?  A. Current object is the object which has currently called the method. The object which is currently in action.  Local variable is a variable which is declared either as a parameter of the function or inside the function itself.  We can make difference between local and instance variable by using this.  We can call constructor using this. If we use this() to call the constructor then that must be the very first statement within constructor otherwise compiler will give error.  Calling constructor using this should always be inside a constructor.  Variable shadowing: When local and instance have same name then local variable hides the instance variable or we can say instance variable is being shadowed by the local variable.\*\*/  /\*\*  \* class Test  {  int b;  int c;    Test()  {  System.out.println("No argument constructor");  b = 1;  c = 1;  }  Test (int b)  {  this(); //calling zero arg constructor using this.  System.out.println("One argument constructor");    this.b = b;  }  Test(int b, int c) //Example of variable of shadowing  {  this(b); //Calling one argument constructor  System.out.println("Two argument constructor");  b = b;  c = c;  }  void fun()  {    //this(); //Error, calling constructor using this should always be inside a constructor.  System.out.println("fun of Test");  System.out.println(this.b);  System.out.println(this.c);  }  }  class StaticDemo  {  public static void main(String args[])  {  Test t1 = new Test(5,10);  Test t2 = new Test(15);  Test t3 = new Test();  t1.fun();  t2.fun();  }  }  \*/ |

# *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) (Object Oriented programming system).
* 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.

### Why use inheritance in java

* For [Method Overriding](https://www.javatpoint.com/method-overriding-in-java) (so [runtime polymorphism](https://www.javatpoint.com/runtime-polymorphism-in-java) can be achieved).
* For Code Reusability.

### Terms used in Inheritance

* **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
* **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
* **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
* **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

### The syntax of Java Inheritance

1. **class** Subclass-name **extends** Superclass-name
2. {
3. //methods and fields
4. }

* The **extends keyword** indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.
* In the terminology of Java, a class which is inherited is called a parent or superclass, and the new class is called child or subclass.

### Java Inheritance Example



As displayed in the above figure, Programmer is the subclass and Employee is the superclass. The relationship between the two classes is **Programmer IS-A Employee**. It means that Programmer is a type of Employee.

1. **class** Employee{
2. **float** salary=40000;
3. }
4. **class** Programmer **extends** Employee{
5. **int** bonus=10000;
6. **public** **static** **void** main(String args[]){
7. Programmer p=**new** Programmer();
8. System.out.println("Programmer salary is:"+p.salary);
9. System.out.println("Bonus of Programmer is:"+p.bonus);
10. }
11. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=Programmer)

Programmer salary is:40000.0

Bonus of programmer is:10000

In the above example, Programmer object can access the field of own class as well as of Employee class i.e. code reusability.

## Types of inheritance in java

* On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.
* In java programming, multiple and hybrid inheritance is supported through interface only. We will learn about interfaces later.



#### Note: Multiple inheritance is not supported in Java through class.

When one class inherits multiple classes, it is known as multiple inheritance. For Example:



## Single Inheritance Example

When a class inherits another class, it is known as a single inheritance. In the example given below, Dog class inherits the Animal class, so there is the single inheritance.

*File: TestInheritance.java*

1. **class** Animal{
2. **void** eat(){System.out.println("eating...");}
3. }
4. **class** Dog **extends** Animal{
5. **void** bark(){System.out.println("barking...");}
6. }
7. **class** TestInheritance{
8. **public** **static** **void** main(String args[]){
9. Dog d=**new** Dog();
10. d.bark();
11. d.eat();
12. }}

Output:

barking...

eating...

## Multilevel Inheritance Example

When there is a chain of inheritance, it is known as multilevel inheritance. As you can see in the example given below, BabyDog class inherits the Dog class which again inherits the Animal class, so there is a multilevel inheritance.

*File: TestInheritance2.java*

1. **class** Animal{
2. **void** eat(){System.out.println("eating...");}
3. }
4. **class** Dog **extends** Animal{
5. **void** bark(){System.out.println("barking...");}
6. }
7. **class** BabyDog **extends** Dog{
8. **void** weep(){System.out.println("weeping...");}
9. }
10. **class** TestInheritance2{
11. **public** **static** **void** main(String args[]){
12. BabyDog d=**new** BabyDog();
13. d.weep();
14. d.bark();
15. d.eat();
16. }}

Output:

weeping...

barking...

eating...

## Hierarchical Inheritance Example

When two or more classes inherits a single class, it is known as hierarchical inheritance. In the example given below, Dog and Cat classes inherits the Animal class, so there is hierarchical inheritance.

*File: TestInheritance3.java*

1. **class** Animal{
2. **void** eat(){System.out.println("eating...");}
3. }
4. **class** Dog **extends** Animal{
5. **void** bark(){System.out.println("barking...");}
6. }
7. **class** Cat **extends** Animal{
8. **void** meow(){System.out.println("meowing...");}
9. }
10. **class** TestInheritance3{
11. **public** **static** **void** main(String args[]){
12. Cat c=**new** Cat();
13. c.meow();
14. c.eat();
15. //c.bark();//C.T.Error
16. }}

Output:

meowing...

eating...

## Q) Why multiple inheritance is not supported in java?

To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.

Since compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error.

1. **class** A{
2. **void** msg(){System.out.println("Hello");}
3. }
4. **class** B{
5. **void** msg(){System.out.println("Welcome");}
6. }
7. **class** C **extends** A,B{//suppose if it were
9. **public** **static** **void** main(String args[]){
10. C obj=**new** C();
11. obj.msg();//Now which msg() method would be invoked?
12. }
13. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=C)

Compile Time Error

|  |
| --- |
| ***===============================Shiv nath sir================================***  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Relationship between Objects\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  Assosciation --> Relationship between Objects  Two types of Assosciation  1)Aggregation - one objec canexist independently without another.Also known as **has-a** relationshi  weak relation between objects  Ex :- car with engine  car required engine  but engine does no reuired car  it is no a strong relation weak relation  so it is known as an aggregration assosciation  2)Composition - Here child object can not exist Independently from the Parent.Also known as an a part-of relationship  Ex : Strong relation betweemn objects  human body with Heart  heart reuires body  body requyires heart it is a strobng relation  it is an composition assosciation  CDAC Employee has Bank Account  class CDACEmployee  {  String bankName;  }  class BankAccount  {  String bsnkName;  }  class AssosciationDemo  {  public static void main(String args[])  {  CDACEmployee emp = new CDACEmployee("Ramesh");  BankAccount bank = new BankAccount("SBI");  System.out.print(emp.empName + " has a bank account in" + bank.bankName )  }  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  Lab work :- Personal details store  make class with personal details\  class PersonalDetails  {  String state;  String mobNo;  String emailId;    void displayPersonalDetail()  {  sop(personalDetails.state);  sop(personalDetails.mobNo);  sop(personalDetails.emailId);  }  }  class Cricketer  {  Personal Details pD;  String name;  String country;  int totmatch;  Cricketer(String n,string c,stri tm,string s,string mn,string email)  pD = new PersonDetails();  pD.state = s;  pd.mobNo = mn;  Pd.email = email;  name = n;  country = c;  totalMatches = tm;  }  void displayDetail()  {  sop(name);  sop(country);  sop(totalMatches);    personalDetails.personalDetails();  }  //Similarly for Employeeeee  class Employee  {    String empId;  String empName;  double salary;  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  IS-A relationship  it is inheritance.iNHERITANCE RELATIONSHIP IS ALSO CLAAED A Is\_a RELATIONSHIP  batsman extends cricketer  we can say  Batsman is a cricketer -->the relation betwen batsman andc cricketeer is inhertance  //Hence inheritance is an IS-A relation  \*\*\*\*\*\*\*\*\*\*\*Composition\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  Composition : wen an object is called in another class it is called composition  It is also called part of relatonship  It is also knowm as Aggregation  Example  class A //  {  b.obj;  }  \*\*\*\*\*\*\*\*Is\_A or Inheritance\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  Inheritance is also known as an IS-A Relation  Example:  class A extends |

# *Aggregation in Java*

* If a class have an entity reference, it is known as Aggregation. Aggregation represents **HAS-A relationship**.
* Consider a situation, Employee object contains many informations such as id, name, emailId etc. It contains one more object named address, which contains its own informations such as city, state, country, zipcode etc. as given below.

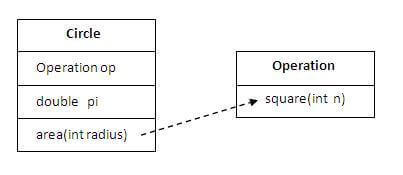
1. **class** Employee{
2. **int** id;
3. String name;
4. Address address;//Address is a class
5. ...
6. }

In such case, Employee has an entity reference address, so relationship is Employee HAS-A address.

### Why use Aggregation?

* For Code Reusability.

### Simple Example of Aggregation



In this example, we have created the reference of Operation class in the Circle class.

1. **class** Operation{
2. **int** square(**int** n){
3. **return** n\*n;
4. }
5. }
7. **class** Circle{
8. Operation op;//aggregation
9. **double** pi=3.14;
11. **double** area(**int** radius){
12. op=**new** Operation();
13. **int** rsquare=op.square(radius);//code reusability (i.e. delegates the method call).
14. **return** pi\*rsquare;
15. }


19. **public** **static** **void** main(String args[]){
20. Circle c=**new** Circle();
21. **double** result=c.area(5);
22. System.out.println(result);
23. }
24. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=Circle)

Output:78.5

### When use Aggregation?

* Code reuse is also best achieved by aggregation when there is no is-a relationship.
* Inheritance should be used only if the relationship is-a is maintained throughout the lifetime of the objects involved; otherwise, aggregation is the best choice.

### Understanding meaningful example of Aggregation

In this example, Employee has an object of Address, address object contains its own informations such as city, state, country etc. In such case relationship is Employee HAS-A address.

#### Address.java

1. **public** **class** Address {
2. String city,state,country;
4. **public** Address(String city, String state, String country) {
5. **this**.city = city;
6. **this**.state = state;
7. **this**.country = country;
8. }
10. }

#### Emp.java

1. **public** **class** Emp {
2. **int** id;
3. String name;
4. Address address;
6. **public** Emp(**int** id, String name,Address address) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.address=address;
10. }
12. **void** display(){
13. System.out.println(id+" "+name);
14. System.out.println(address.city+" "+address.state+" "+address.country);
15. }
17. **public** **static** **void** main(String[] args) {
18. Address address1=**new** Address("gzb","UP","india");
19. Address address2=**new** Address("gno","UP","india");
21. Emp e=**new** Emp(111,"varun",address1);
22. Emp e2=**new** Emp(112,"arun",address2);
24. e.display();
25. e2.display();
27. }
28. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=Emp)

Output:111 varun

gzb UP india

112 arun

gno UP india

***Java Polymorphism***

# Method Overloading in Java

* If a [class](https://www.javatpoint.com/object-and-class-in-java) has multiple methods having same name but different in parameters, it is known as **Method Overloading**.
* If we have to perform only one operation, having same name of the methods increases the readability of the [program](https://www.javatpoint.com/java-programs).
* Suppose you have to perform addition of the given numbers but there can be any number of arguments, if you write the method such as a(int,int) for two parameters, and b(int,int,int) for three parameters then it may be difficult for you as well as other programmers to understand the behavior of the method because its name differs.

So, we perform method overloading to figure out the program quickly.

## Advantage of method overloading

Method overloading increases the readability of the program.

### Different ways to overload the method

**There are two ways to overload the method in java**

* By changing number of arguments
* By changing the data type

#### In Java, Method Overloading is not possible by changing the return type of the method only.

### 1) Method Overloading: changing no. of arguments

In this example, we have created two methods, first add() method performs addition of two numbers and second add method performs addition of three numbers.

In this example, we are creating [static methods](https://www.javatpoint.com/static-keyword-in-java) so that we don't need to create instance for calling methods.

1. **class** Adder{
2. **static** **int** add(**int** a,**int** b){**return** a+b;}
3. **static** **int** add(**int** a,**int** b,**int** c){**return** a+b+c;}
4. }
5. **class** TestOverloading1{
6. **public** **static** **void** main(String[] args){
7. System.out.println(Adder.add(11,11));
8. System.out.println(Adder.add(11,11,11));
9. }}

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestOverloading1)

Output:

22

33

### 2) Method Overloading: changing data type of arguments

In this example, we have created two methods that differs in [data type](https://www.javatpoint.com/java-data-types). The first add method receives two integer arguments and second add method receives two double arguments.

1. **class** Adder{
2. **static** **int** add(**int** a, **int** b){**return** a+b;}
3. **static** **double** add(**double** a, **double** b){**return** a+b;}
4. }
5. **class** TestOverloading2{
6. **public** **static** **void** main(String[] args){
7. System.out.println(Adder.add(11,11));
8. System.out.println(Adder.add(12.3,12.6));
9. }}

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestOverloading2)

Output:

22

24.9

### Q) Why Method Overloading is not possible by changing the return type of method only?

In java, method overloading is not possible by changing the return type of the method only because of ambiguity. Let's see how ambiguity may occur:

1. **class** Adder{
2. **static** **int** add(**int** a,**int** b){**return** a+b;}
3. **static** **double** add(**int** a,**int** b){**return** a+b;}
4. }
5. **class** TestOverloading3{
6. **public** **static** **void** main(String[] args){
7. System.out.println(Adder.add(11,11));//ambiguity
8. }}

Output:

Compile Time Error: method add(int,int) is already defined in class Adder

System.out.println(Adder.add(11,11)); //Here, how can java determine which sum() method should be called?

#### Note: Compile Time Error is better than Run Time Error. So, java compiler renders compiler time error if you declare the same method having same parameters.

### Can we overload java main() method?

Yes, by method overloading. You can have any number of main methods in a class by method overloading. But [JVM](https://www.javatpoint.com/jvm-java-virtual-machine) calls main() method which receives string array as arguments only. Let's see the simple example:

1. **class** TestOverloading4{
2. **public** **static** **void** main(String[] args){System.out.println("main with String[]");}
3. **public** **static** **void** main(String args){System.out.println("main with String");}
4. **public** **static** **void** main(){System.out.println("main without args");}
5. }

Output:

main with String[]

## Method Overloading and Type Promotion

One type is promoted to another implicitly if no matching datatype is found. Let's understand the concept by the figure given below:



As displayed in the above diagram, byte can be promoted to short, int, long, float or double. The short datatype can be promoted to int, long, float or double. The char datatype can be promoted to int,long,float or double and so on.

### Example of Method Overloading with TypePromotion

1. **class** OverloadingCalculation1{
2. **void** sum(**int** a,**long** b){System.out.println(a+b);}
3. **void** sum(**int** a,**int** b,**int** c){System.out.println(a+b+c);}
5. **public** **static** **void** main(String args[]){
6. OverloadingCalculation1 obj=**new** OverloadingCalculation1();
7. obj.sum(20,20);//now second int literal will be promoted to long
8. obj.sum(20,20,20);
10. }
11. }

Output:40

60

### Example of Method Overloading with Type Promotion if matching found

If there are matching type arguments in the method, type promotion is not performed.

1. **class** OverloadingCalculation2{
2. **void** sum(**int** a,**int** b){System.out.println("int arg method invoked");}
3. **void** sum(**long** a,**long** b){System.out.println("long arg method invoked");}
5. **public** **static** **void** main(String args[]){
6. OverloadingCalculation2 obj=**new** OverloadingCalculation2();
7. obj.sum(20,20);//now int arg sum() method gets invoked
8. }
9. }

Output:int arg method invoked

### Example of Method Overloading with Type Promotion in case of ambiguity

If there are no matching type arguments in the method, and each method promotes similar number of arguments, there will be ambiguity.

1. **class** OverloadingCalculation3{
2. **void** sum(**int** a,**long** b){System.out.println("a method invoked");}
3. **void** sum(**long** a,**int** b){System.out.println("b method invoked");}
5. **public** **static** **void** main(String args[]){
6. OverloadingCalculation3 obj=**new** OverloadingCalculation3();
7. obj.sum(20,20);//now ambiguity
8. }
9. }

Output:Compile Time Error

#### One type is not de-promoted implicitly for example double cannot be depromoted to any type implicitly.

# Method Overriding in Java

If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java**.

In other words, If a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding.

Usage of Java Method Overriding

* Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
* Method overriding is used for runtime polymorphism

Rules for Java Method Overriding

1. The method must have the same name as in the parent class
2. The method must have the same parameter as in the parent class.
3. There must be an IS-A relationship (inheritance).



Understanding the problem without method overriding

Let's understand the problem that we may face in the program if we don't use method overriding.

1. //Java Program to demonstrate why we need method overriding
2. //Here, we are calling the method of parent class with child
3. //class object.
4. //Creating a parent class
5. **class** Vehicle{
6. **void** run(){System.out.println("Vehicle is running");}
7. }
8. //Creating a child class
9. **class** Bike **extends** Vehicle{
10. **public** **static** **void** main(String args[]){
11. //creating an instance of child class
12. Bike obj = **new** Bike();
13. //calling the method with child class instance
14. obj.run();
15. }
16. }

Output:

Vehicle is running

Problem is that I have to provide a specific implementation of run() method in subclass that is why we use method overriding.

Example of method overriding

In this example, we have defined the run method in the subclass as defined in the parent class but it has some specific implementation. The name and parameter of the method are the same, and there is IS-A relationship between the classes, so there is method overriding.

1. //Java Program to illustrate the use of Java Method Overriding
2. //Creating a parent class.
3. **class** Vehicle{
4. //defining a method
5. **void** run(){System.out.println("Vehicle is running");}
6. }
7. //Creating a child class
8. **class** Bike2 **extends** Vehicle{
9. //defining the same method as in the parent class
10. **void** run(){System.out.println("Bike is running safely");}
12. **public** **static** **void** main(String args[]){
13. Bike2 obj = **new** Bike2();//creating object
14. obj.run();//calling method
15. }
16. }

Output:

Bike is running safely

A real example of Java Method Overriding

Consider a scenario where Bank is a class that provides functionality to get the rate of interest. However, the rate of interest varies according to banks. For example, SBI, ICICI and AXIS banks could provide 8%, 7%, and 9% rate of interest.



Java method overriding is mostly used in Runtime Polymorphism which we will learn in next pages.

1. //Java Program to demonstrate the real scenario of Java Method Overriding
2. //where three classes are overriding the method of a parent class.
3. //Creating a parent class.
4. **class** Bank{
5. **int** getRateOfInterest(){**return** 0;}
6. }
7. //Creating child classes.
8. **class** SBI **extends** Bank{
9. **int** getRateOfInterest(){**return** 8;}
10. }
12. **class** ICICI **extends** Bank{
13. **int** getRateOfInterest(){**return** 7;}
14. }
15. **class** AXIS **extends** Bank{
16. **int** getRateOfInterest(){**return** 9;}
17. }
18. //Test class to create objects and call the methods
19. **class** Test2{
20. **public** **static** **void** main(String args[]){
21. SBI s=**new** SBI();
22. ICICI i=**new** ICICI();
23. AXIS a=**new** AXIS();
24. System.out.println("SBI Rate of Interest: "+s.getRateOfInterest());
25. System.out.println("ICICI Rate of Interest: "+i.getRateOfInterest());
26. System.out.println("AXIS Rate of Interest: "+a.getRateOfInterest());
27. }
28. }

Output:

SBI Rate of Interest: 8

ICICI Rate of Interest: 7

AXIS Rate of Interest: 9

Can we override static method?

No, a static method cannot be overridden. It can be proved by runtime polymorphism.

Why can we not override static method?

It is because the static method is bound with class whereas instance method is bound with an object. Static belongs to the class area, and an instance belongs to the heap area.

Can we override java main method?

No, because the main is a static method.

# Covariant Return Type

The covariant return type specifies that the return type may vary in the same direction as the subclass.

Before Java5, it was not possible to override any method by changing the return type. But now, since Java5, it is possible to override method by changing the return type if subclass overrides any method whose return type is Non-Primitive but it changes its return type to subclass type. Let's take a simple example:

#### Note: If you are beginner to java, skip this topic and return to it after OOPs concepts.

### Simple example of Covariant Return Type

1. **class** A{
2. A get(){**return** **this**;}
3. }
5. **class** B1 **extends** A{
6. B1 get(){**return** **this**;}
7. **void** message(){System.out.println("welcome to covariant return type");}
9. **public** **static** **void** main(String args[]){
10. **new** B1().get().message();
11. }
12. }

Output:welcome to covariant return type

As you can see in the above example, the return type of the get() method of A class is A but the return type of the get() method of B class is B. Both methods have different return type but it is method overriding. This is known as covariant return type.

### How is Covariant return types implemented?

Java doesn't allow the return type based overloading but JVM always allows return type based overloading. JVM uses full signature of a method for lookup/resolution. Full signature means it includes return type in addition to argument types. i.e., a class can have two or more methods differing only by return type. javac uses this fact to implement covariant return types.

# *Super Keyword in Java*

* The **super** keyword in Java is a reference variable which is used to refer immediate parent class object.
* Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

## Usage of Java super Keyword

* super can be used to refer immediate parent class instance variable.
* super can be used to invoke immediate parent class method.
* super() can be used to invoke immediate parent class constructor.



## 1) super is used to refer immediate parent class instance variable.

We can use super keyword to access the data member or field of parent class. It is used if parent class and child class have same fields.

1. **class** Animal{
2. String color="white";
3. }
4. **class** Dog **extends** Animal{
5. String color="black";
6. **void** printColor(){
7. System.out.println(color);//prints color of Dog class
8. System.out.println(**super**.color);//prints color of Animal class
9. }
10. }
11. **class** TestSuper1{
12. **public** **static** **void** main(String args[]){
13. Dog d=**new** Dog();
14. d.printColor();
15. }}

Output:

black

white

* In the above example, Animal and Dog both classes have a common property color. If we print color property, it will print the color of current class by default. To access the parent property, we need to use super keyword.

## 2) super can be used to invoke parent class method

The super keyword can also be used to invoke parent class method. It should be used if subclass contains the same method as parent class. In other words, it is used if method is overridden.

1. **class** Animal{
2. **void** eat(){System.out.println("eating...");}
3. }
4. **class** Dog **extends** Animal{
5. **void** eat(){System.out.println("eating bread...");}
6. **void** bark(){System.out.println("barking...");}
7. **void** work(){
8. **super**.eat();
9. bark();
10. }
11. }
12. **class** TestSuper2{
13. **public** **static** **void** main(String args[]){
14. Dog d=**new** Dog();
15. d.work();
16. }}

Output:

eating...

barking...

* In the above example Animal and Dog both classes have eat() method if we call eat() method from Dog class, it will call the eat() method of Dog class by default because priority is given to local.
* To call the parent class method, we need to use super keyword.

## 3) super is used to invoke parent class constructor.

The super keyword can also be used to invoke the parent class constructor. Let's see a simple example:

1. **class** Animal{
2. Animal(){System.out.println("animal is created");}
3. }
4. **class** Dog **extends** Animal{
5. Dog(){
6. **super**();
7. System.out.println("dog is created");
8. }
9. }
10. **class** TestSuper3{
11. **public** **static** **void** main(String args[]){
12. Dog d=**new** Dog();
13. }}

Output:

animal is created

dog is created

**Note: super() is added in each class constructor automatically by compiler if there is no super() or this().**



As we know well that default constructor is provided by compiler automatically if there is no constructor. But, it also adds super() as the first statement.

**Another example of super keyword where super() is provided by the compiler implicitly.**

1. **class** Animal{
2. Animal(){System.out.println("animal is created");}
3. }
4. **class** Dog **extends** Animal{
5. Dog(){
6. System.out.println("dog is created");
7. }
8. }
9. **class** TestSuper4{
10. **public** **static** **void** main(String args[]){
11. Dog d=**new** Dog();
12. }}

Output:

animal is created

dog is created

## super example: real use

Let's see the real use of super keyword. Here, Emp class inherits Person class so all the properties of Person will be inherited to Emp by default. To initialize all the property, we are using parent class constructor from child class. In such way, we are reusing the parent class constructor.

1. **class** Person{
2. **int** id;
3. String name;
4. Person(**int** id,String name){
5. **this**.id=id;
6. **this**.name=name;
7. }
8. }
9. **class** Emp **extends** Person{
10. **float** salary;
11. Emp(**int** id,String name,**float** salary){
12. **super**(id,name);//reusing parent constructor
13. **this**.salary=salary;
14. }
15. **void** display(){System.out.println(id+" "+name+" "+salary);}
16. }
17. **class** TestSuper5{
18. **public** **static** **void** main(String[] args){
19. Emp e1=**new** Emp(1,"ankit",45000f);
20. e1.display();
21. }}

Output:

1 ankit 45000

------------------------------------------------------------------------------------------------------------------------------

**Instance initializer block**

* Instance Initializer block is used to initialize the instance data member. It run each time when object of the class is created.
* The initialization of the instance variable can be done directly but there can be performed extra operations while initializing the instance variable in the instance initializer block.

Que) What is the use of instance initializer block while we can directly assign a value in instance data member? For example:

class Bike{

int speed=100;

}

Rules for instance initializer block :

1. The instance initializer block is created when instance of the class is created.
2. The instance initializer block is invoked after the parent class constructor is invoked (i.e. after super() constructor call).
3. The instance initializer block comes in the order in which they appear.

# *Final Keyword In Java*

The **final keyword** in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

1. variable
2. method
3. class

The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only. We will have detailed learning of these. Let's first learn the basics of final keyword.



## 1) Java final variable

If you make any variable as final, you cannot change the value of final variable(It will be constant).

### Example of final variable

There is a final variable speedlimit, we are going to change the value of this variable, but It can't be changed because final variable once assigned a value can never be changed.

1. **class** Bike9{
2. **final** **int** speedlimit=90;//final variable
3. **void** run(){
4. speedlimit=400;
5. }
6. **public** **static** **void** main(String args[]){
7. Bike9 obj=**new**  Bike9();
8. obj.run();
9. }
10. }//end of class

Output:

Compile Time Error

## 2) Java final method

If you make any method as final, you cannot override it.

### Example of final method

1. **class** Bike{
2. **final** **void** run(){System.out.println("running");}
3. }
5. **class** Honda **extends** Bike{
6. **void** run(){System.out.println("running safely with 100kmph");}
8. **public** **static** **void** main(String args[]){
9. Honda honda= **new** Honda();
10. honda.run();
11. }
12. }

Output:Compile Time Error

## 3) Java final class

If you make any class as final, you cannot extend it.

### Example of final class

1. **final** **class** Bike{}
3. **class** Honda1 **extends** Bike{
4. **void** run(){System.out.println("running safely with 100kmph");}
6. **public** **static** **void** main(String args[]){
7. Honda1 honda= **new** Honda1();
8. honda.run();
9. }
10. }

Output:Compile Time Error

### Q) Is final method inherited?

Ans) Yes, final method is inherited but you cannot override it. For Example:

1. **class** Bike{
2. **final** **void** run(){System.out.println("running...");}
3. }
4. **class** Honda2 **extends** Bike{
5. **public** **static** **void** main(String args[]){
6. **new** Honda2().run();
7. }
8. }

Output:running...

### Q) What is blank or uninitialized final variable?

* A final variable that is not initialized at the time of declaration is known as blank final variable.
* If you want to create a variable that is initialized at the time of creating object and once initialized may not be changed, it is useful. For example PAN CARD number of an employee.

It can be initialized only in constructor.

### Example of blank final variable

1. **class** Student{
2. **int** id;
3. String name;
4. **final** String PAN\_CARD\_NUMBER;
5. ...
6. }

### Que) Can we initialize blank final variable?

Yes, but only in constructor. For example:

1. **class** Bike10{
2. **final** **int** speedlimit;//blank final variable
4. Bike10(){
5. speedlimit=70;
6. System.out.println(speedlimit);
7. }
9. **public** **static** **void** main(String args[]){
10. **new** Bike10();
11. }
12. }

Output: 70

### static blank final variable

A static final variable that is not initialized at the time of declaration is known as static blank final variable. It can be initialized only in static block.

### Example of static blank final variable

1. **class** A{
2. **static** **final** **int** data;//static blank final variable
3. **static**{ data=50;}
4. **public** **static** **void** main(String args[]){
5. System.out.println(A.data);
6. }
7. }

### Q) What is final parameter?

If you declare any parameter as final, you cannot change the value of it.

1. **class** Bike11{
2. **int** cube(**final** **int** n){
3. n=n+2;//can't be changed as n is final
4. n\*n\*n;
5. }
6. **public** **static** **void** main(String args[]){
7. Bike11 b=**new** Bike11();
8. b.cube(5);
9. }
10. }

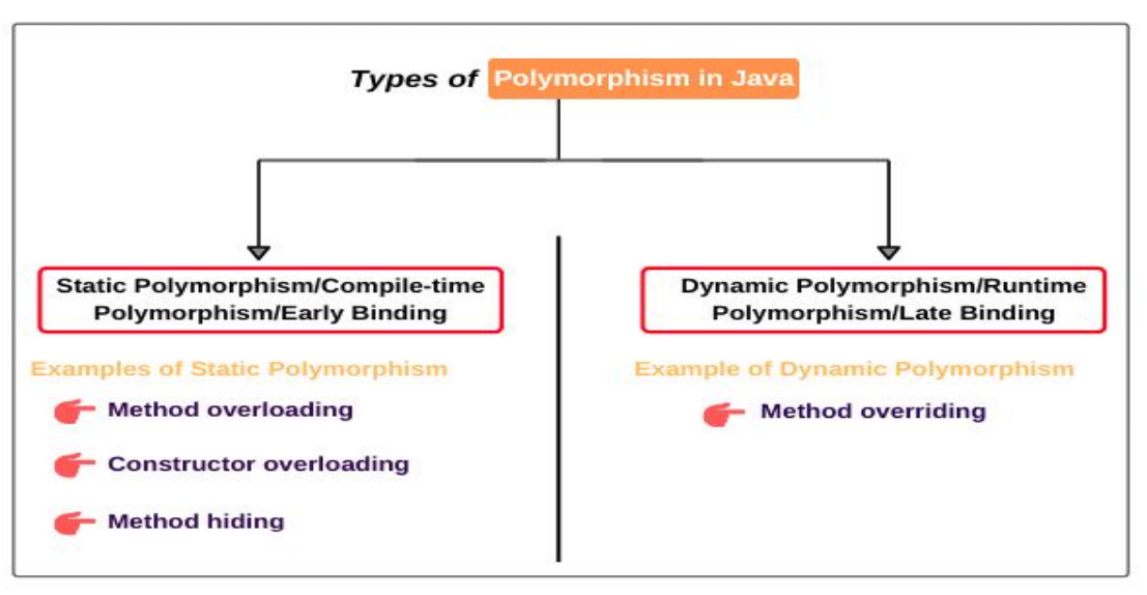
Output: Compile Time Error

### Q) Can we declare a constructor final?

No, because constructor is never inherited.

# *Polymorphism in Java*

* **Polymorphism in Java** is a concept by which we can perform a single action in different ways. Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.
* There are two types of polymorphism in Java: compile-time polymorphism and runtime polymorphism. We can perform polymorphism in java by method overloading and method overriding.
* If you overload a static method in Java, it is the example of compile time polymorphism. Here, we will focus on runtime polymorphism in java.



Runtime Polymorphism in Java

* **Runtime polymorphism** or **Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time.
* In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.
* Let's first understand the upcasting before Runtime Polymorphism.

Upcasting

* If the reference variable of Parent class refers to the object of Child class, it is known as upcasting. For example:



1. **class** A{}
2. **class** B **extends** A{}
3. A a=**new** B();//upcasting

For upcasting, we can use the reference variable of class type or an interface type. For Example:

1. **interface** I{}
2. **class** A{}
3. **class** B **extends** A **implements** I{}

Here, the relationship of B class would be:

B IS-A A

B IS-A I

B IS-A Object

Since Object is the root class of all classes in Java, so we can write B IS-A Object.

Example of Java Runtime Polymorphism

In this example, we are creating two classes Bike and Splendor. Splendor class extends Bike class and overrides its run() method. We are calling the run method by the reference variable of Parent class. Since it refers to the subclass object and subclass method overrides the Parent class method, the subclass method is invoked at runtime.

Since method invocation is determined by the JVM not compiler, it is known as runtime polymorphism.

1. **class** Bike{
2. **void** run(){System.out.println("running");}
3. }
4. **class** Splendor **extends** Bike{
5. **void** run(){System.out.println("running safely with 60km");}
7. **public** **static** **void** main(String args[]){
8. Bike b = **new** Splendor();//upcasting
9. b.run();
10. }
11. }

Output:

running safely with 60km.

Java Runtime Polymorphism Example: Bank

Consider a scenario where Bank is a class that provides a method to get the rate of interest. However, the rate of interest may differ according to banks. For example, SBI, ICICI, and AXIS banks are providing 8.4%, 7.3%, and 9.7% rate of interest.



Note: This example is also given in method overriding but there was no upcasting.

1. **class** Bank{
2. **float** getRateOfInterest(){**return** 0;}
3. }
4. **class** SBI **extends** Bank{
5. **float** getRateOfInterest(){**return** 8.4f;}
6. }
7. **class** ICICI **extends** Bank{
8. **float** getRateOfInterest(){**return** 7.3f;}
9. }
10. **class** AXIS **extends** Bank{
11. **float** getRateOfInterest(){**return** 9.7f;}
12. }
13. **class** TestPolymorphism{
14. **public** **static** **void** main(String args[]){
15. Bank b;
16. b=**new** SBI();
17. System.out.println("SBI Rate of Interest: "+b.getRateOfInterest());
18. b=**new** ICICI();
19. System.out.println("ICICI Rate of Interest: "+b.getRateOfInterest());
20. b=**new** AXIS();
21. System.out.println("AXIS Rate of Interest: "+b.getRateOfInterest());
22. }
23. }

Output:

SBI Rate of Interest: 8.4

ICICI Rate of Interest: 7.3

AXIS Rate of Interest: 9.7

Java Runtime Polymorphism Example: Shape

1. **class** Shape{
2. **void** draw(){System.out.println("drawing...");}
3. }
4. **class** Rectangle **extends** Shape{
5. **void** draw(){System.out.println("drawing rectangle...");}
6. }
7. **class** Circle **extends** Shape{
8. **void** draw(){System.out.println("drawing circle...");}
9. }
10. **class** Triangle **extends** Shape{
11. **void** draw(){System.out.println("drawing triangle...");}
12. }
13. **class** TestPolymorphism2{
14. **public** **static** **void** main(String args[]){
15. Shape s;
16. s=**new** Rectangle();
17. s.draw();
18. s=**new** Circle();
19. s.draw();
20. s=**new** Triangle();
21. s.draw();
22. }
23. }

Output:

drawing rectangle...

drawing circle...

drawing triangle...

Java Runtime Polymorphism Example: Animal

1. **class** Animal{
2. **void** eat(){System.out.println("eating...");}
3. }
4. **class** Dog **extends** Animal{
5. **void** eat(){System.out.println("eating bread...");}
6. }
7. **class** Cat **extends** Animal{
8. **void** eat(){System.out.println("eating rat...");}
9. }
10. **class** Lion **extends** Animal{
11. **void** eat(){System.out.println("eating meat...");}
12. }
13. **class** TestPolymorphism3{
14. **public** **static** **void** main(String[] args){
15. Animal a;
16. a=**new** Dog();
17. a.eat();
18. a=**new** Cat();
19. a.eat();
20. a=**new** Lion();
21. a.eat();
22. }}

Output:

eating bread...

eating rat...

eating meat...

Java Runtime Polymorphism with Data Member

* A method is overridden, not the data members, so runtime polymorphism can't be achieved by data members.
* In the example given below, both the classes have a data member speedlimit. We are accessing the data member by the reference variable of Parent class which refers to the subclass object. Since we are accessing the data member which is not overridden, hence it will access the data member of the Parent class always.

Rule: Runtime polymorphism can't be achieved by data members.

1. **class** Bike{
2. **int** speedlimit=90;
3. }
4. **class** Honda3 **extends** Bike{
5. **int** speedlimit=150;
7. **public** **static** **void** main(String args[]){
8. Bike obj=**new** Honda3();
9. System.out.println(obj.speedlimit);//90
10. }

Output:

90

Java Runtime Polymorphism with Multilevel Inheritance

Let's see the simple example of Runtime Polymorphism with multilevel inheritance.

1. **class** Animal{
2. **void** eat(){System.out.println("eating");}
3. }
4. **class** Dog **extends** Animal{
5. **void** eat(){System.out.println("eating fruits");}
6. }
7. **class** BabyDog **extends** Dog{
8. **void** eat(){System.out.println("drinking milk");}
9. **public** **static** **void** main(String args[]){
10. Animal a1,a2,a3;
11. a1=**new** Animal();
12. a2=**new** Dog();
13. a3=**new** BabyDog();
14. a1.eat();
15. a2.eat();
16. a3.eat();
17. }
18. }

Output:

eating

eating fruits

drinking Milk

Try for Output

1. **class** Animal{
2. **void** eat(){System.out.println("animal is eating...");}
3. }
4. **class** Dog **extends** Animal{
5. **void** eat(){System.out.println("dog is eating...");}
6. }
7. **class** BabyDog1 **extends** Dog{
8. **public** **static** **void** main(String args[]){
9. Animal a=**new** BabyDog1();
10. a.eat();
11. }}

Output:

Dog is eating

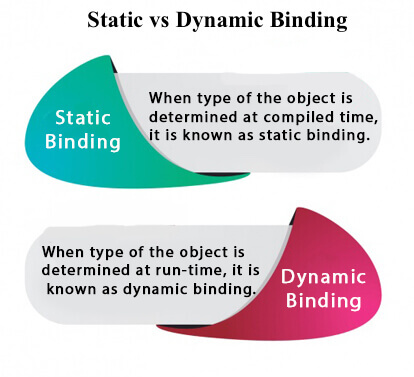
Since, BabyDog is not overriding the eat() method, so eat() method of Dog class is invoked.

***Static Binding and Dynamic Binding***

Connecting a method call to the method body is known as binding.

There are two types of binding

1. **Static Binding (also known as Early Binding).**
2. **Dynamic Binding (also known as Late Binding).**



Understanding Type

Let's understand the type of instance.

1) variables have a type

Each variable has a type, it may be primitive and non-primitive.

1. **int** data=30;

Here data variable is a type of int.

2) References have a type

1. **class** Dog{
2. **public** **static** **void** main(String args[]){
3. Dog d1;//Here d1 is a type of Dog
4. }
5. }

3) Objects have a type

|  |
| --- |
| An object is an instance of particular java class,but it is also an instance of its superclass. |

1. **class** Animal{}
3. **class** Dog **extends** Animal{
4. **public** **static** **void** main(String args[]){
5. Dog d1=**new** Dog();
6. }
7. }

|  |
| --- |
| Here d1 is an instance of Dog class, but it is also an instance of Animal. |

static binding

When type of the object is determined at compiled time(by the compiler), it is known as static binding.

If there is any private, final or static method in a class, there is static binding.

Example of static binding

1. **class** Dog{
2. **private** **void** eat(){System.out.println("dog is eating...");}
4. **public** **static** **void** main(String args[]){
5. Dog d1=**new** Dog();
6. d1.eat();
7. }
8. }

Dynamic binding

When type of the object is determined at run-time, it is known as dynamic binding.

Example of dynamic binding

1. **class** Animal{
2. **void** eat(){System.out.println("animal is eating...");}
3. }
5. **class** Dog **extends** Animal{
6. **void** eat(){System.out.println("dog is eating...");}
8. **public** **static** **void** main(String args[]){
9. Animal a=**new** Dog();
10. a.eat();
11. }
12. }

Output:dog is eating...

|  |
| --- |
| In the above example object type cannot be determined by the compiler, because the instance of  Dog is also an instance of Animal.So compiler doesn't know its type, only its base type. |

|  |
| --- |
| ***===================================Shiv nath sir=============================***  +-\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Binding\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Linking Between Calling Method and Called Method.whenever we make a call to a function,which function will be called ,deciding that thing is called binding.fun() --> Calling methodvoid fun() --> Called MethodStatic Binding --> Binding at Compile TimeDyanmic Binding --> Binding at Run TimeIn the case overriding method call,dynamic binding happensStatic Binding -> Compiler takes decision (or does binding ) based on the refrence type  Dynamic Binding -> JVM (at run time) takes decision(or does binding) based on the type of the refrenced object(i.enum . the object whih is being refrenced). Exmapleclass First{ void fun() { } void anotherfun(int x) { } void anotherfun() { } void anotherFun() { }}class Second extends First { void fun() { } void difffun() { } }class Third extends First { void fun() { } void difffun() { } }Class BindingDemo{ public static void main(String args[]) { First f = new First(); f.fun(); //Static Binding f = new Second(); f.fun(); //Dynamic Binding f.diffFun();  f.anotherFun();  f = new Second();  f.fun(); //Dynamic Binding  f.diffFun();  f.anotherFun();  }  } |
|  |

# *Java instanceof*

* The **java instanceof operator** is used to test whether the object is an instance of the specified type (class or subclass or interface).
* The instanceof in java is also known as type *comparison operator* because it compares the instance with type. It returns either true or false. If we apply the instanceof operator with any variable that has null value, it returns false.

### Simple example of java instanceof

Let's see the simple example of instance operator where it tests the current class.

1. **class** Simple1{
2. **public** **static** **void** main(String args[]){
3. Simple1 s=**new** Simple1();
4. System.out.println(s **instanceof** Simple1);//true
5. }
6. }

Output:true

An object of subclass type is also a type of parent class. For example, if Dog extends Animal then object of Dog can be referred by either Dog or Animal class.

## Another example of java instanceof operator

1. **class** Animal{}
2. **class** Dog1 **extends** Animal{//Dog inherits Animal
4. **public** **static** **void** main(String args[]){
5. Dog1 d=**new** Dog1();
6. System.out.println(d **instanceof** Animal);//true
7. }
8. }

Output:true

## instanceof in java with a variable that have null value

If we apply instanceof operator with a variable that have null value, it returns false. Let's see the example given below where we apply instanceof operator with the variable that have null value.

1. **class** Dog2{
2. **public** **static** **void** main(String args[]){
3. Dog2 d=**null**;
4. System.out.println(d **instanceof** Dog2);//false
5. }
6. }

Output:false

## Downcasting with java instanceof operator

When Subclass type refers to the object of Parent class, it is known as downcasting. If we perform it directly, compiler gives Compilation error. If you perform it by typecasting, ClassCastException is thrown at runtime. But if we use instanceof operator, downcasting is possible.

1. Dog d=**new** Animal();//Compilation error

If we perform downcasting by typecasting, ClassCastException is thrown at runtime.

1. Dog d=(Dog)**new** Animal();
2. //Compiles successfully but ClassCastException is thrown at runtime

### Possibility of downcasting with instanceof

Let's see the example, where downcasting is possible by instanceof operator.

1. **class** Animal { }
3. **class** Dog3 **extends** Animal {
4. **static** **void** method(Animal a) {
5. **if**(a **instanceof** Dog3){
6. Dog3 d=(Dog3)a;//downcasting
7. System.out.println("ok downcasting performed");
8. }
9. }
11. **public** **static** **void** main (String [] args) {
12. Animal a=**new** Dog3();
13. Dog3.method(a);
14. }
16. }

Output:ok downcasting performed

### Downcasting without the use of java instanceof

Downcasting can also be performed without the use of instanceof operator as displayed in the following example:

1. **class** Animal { }
2. **class** Dog4 **extends** Animal {
3. **static** **void** method(Animal a) {
4. Dog4 d=(Dog4)a;//downcasting
5. System.out.println("ok downcasting performed");
6. }
7. **public** **static** **void** main (String [] args) {
8. Animal a=**new** Dog4();
9. Dog4.method(a);
10. }
11. }

Output:ok downcasting performed

Let's take closer look at this, actual object that is referred by a, is an object of Dog class. So if we downcast it, it is fine. But what will happen if we write:

1. Animal a=**new** Animal();
2. Dog.method(a);
3. //Now ClassCastException but not in case of instanceof operator

### Understanding Real use of instanceof in java

Let's see the real use of instanceof keyword by the example given below.

1. **interface** Printable{}
2. **class** A **implements** Printable{
3. **public** **void** a(){System.out.println("a method");}
4. }
5. **class** B **implements** Printable{
6. **public** **void** b(){System.out.println("b method");}
7. }
9. **class** Call{
10. **void** invoke(Printable p){//upcasting
11. **if**(p **instanceof** A){
12. A a=(A)p;//Downcasting
13. a.a();
14. }
15. **if**(p **instanceof** B){
16. B b=(B)p;//Downcasting
17. b.b();
18. }
20. }
21. }//end of Call class
23. **class** Test4{
24. **public** **static** **void** main(String args[]){
25. Printable p=**new** B();
26. Call c=**new** Call();
27. c.invoke(p);
28. }
29. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=Test4)

Output: b method

***Abstraction in Java***

|  |
| --- |
| ***================================Shiv nath sir=============================***  /\*Abstract Method and Classes  Abstract is a Keywrd  This keyword can be used in Method and Classes  Abstract Method  Methods which does not have any implementation or body  Must be overriden inside child cass  Abstract Class  can not be instaniated .means we can not create any object of abstract class  Note:  1)If we have any Abstract Method then class must be declare as an abstract  2)If we have an Abstract Class then method will be or not abstract it is not complusory  \*/  /\*  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Abstract in Method or Function\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  class Cricketer  {  String name;  String Country;  int totalMatches;    void dsiplayDetail()  {  //Print name,country and total matches  }    abstract void calculateAverage(); //Cricketer class want this method must exist in all of child class  //Avearage should be in all of its child class  }  class Batsman extends Cricketer  {  int totalRun;  void calculateAverage() //Forcing the child to Calculate Average  //Wih means overriding the child or forcing the child to claculate Average  {  double avg = totalRun/totalMatches;  System.out.println(avg);  }  }  class Bowler extends Cricketer  {  int totalWickets;  void calculateAverage()  {  double avg = totalWickets/totalMatches;  System.out.println(avg);  }  }  class CricketerDemo  {  public static void main(String args[])  {    }  }  Abstract class it will have data but it will not use the data it will give to its child\  i.e if we abstract in class then no object can be created and use the class  but it will give all details to the childs  ex ; cricketer is an abstract class no object wll be created it will give its data to child like variable and printdetails  batsman and bowler is an child it will use all of data cricketer abstract class  like printdetails and all  Constructor in Abstract -- Add these points from sir notes  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Abstract in Class\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  abstract class Cricketer //This class is not instantied  {  String name;  String Country;  int totalMatches;    void dsiplayDetail()  {  //Print name,country and total matches  }    void calculateAverage(); //Cricketer class want this method must exist in all of child class  //Avearage should be in all of its child class  }  class Batsman extends Cricketer  {  int totalRun;  void calculateAverage() //Forcing the child to Calculate Average  //Wih means overriding the child or forcing the child to claculate Average  {  double avg = totalRun/totalMatches;  System.out.println(avg);  }  }  class Bowler extends Cricketer  {  int totalWickets;  void calculateAverage()  {  double avg = totalWickets/totalMatches;  System.out.println(avg);  }  }  class CricketerDemo  {  public static void main(String args[])  {  //The below staement will give error because we cannot create of an abstract class cricketer  new cricketer(); //we cannot create object of an cricketer as it is an Abstract Class  }  }  \*/ |

***Abstract class in Java***

* A class which is declared with the abstract keyword is known as an abstract class in [Java](https://www.javatpoint.com/java-tutorial). It can have abstract and non-abstract methods (method with the body).
* Before learning the Java abstract class, let's understand the abstraction in Java first.

***Abstraction in Java***

* **Abstraction** is a process of hiding the implementation details and showing only functionality to the user.
* Another way, it shows only essential things to the user and hides the internal details, for example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.
* Abstraction lets you focus on what the [object](https://www.javatpoint.com/object-and-class-in-java) does instead of how it does it.

Ways to achieve Abstraction

There are two ways to achieve abstraction in java

1. Abstract class (0 to 100%)
2. Interface (100%)

**Abstract class in Java**

A class which is declared as abstract is known as an **abstract class**. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.

Points to Remember

* An abstract class must be declared with an abstract keyword.
* It can have abstract and non-abstract methods.
* It cannot be instantiated.
* It can have [constructors](https://www.javatpoint.com/java-constructor) and static methods also.
* It can have final methods which will force the subclass not to change the body of the method.



**Example of abstract class**

1. **abstract** **class** A{}

Abstract Method in Java

A method which is declared as abstract and does not have implementation is known as an abstract method.

**Example of abstract method**

1. **abstract** **void** printStatus();//no method body and abstract

Example of Abstract class that has an abstract method

In this example, Bike is an abstract class that contains only one abstract method run. Its implementation is provided by the Honda class.

1. **abstract** **class** Bike{
2. **abstract** **void** run();
3. }
4. **class** Honda4 **extends** Bike{
5. **void** run(){System.out.println("running safely");}
6. **public** **static** **void** main(String args[]){
7. Bike obj = **new** Honda4();
8. obj.run();
9. }
10. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=Honda4)

running safely

Understanding the real scenario of Abstract class

In this example, Shape is the abstract class, and its implementation is provided by the Rectangle and Circle classes.

Mostly, we don't know about the implementation class (which is hidden to the end user), and an object of the implementation class is provided by the **factory method**.

A **factory method** is a method that returns the instance of the class. We will learn about the factory method later.

In this example, if you create the instance of Rectangle class, draw() method of Rectangle class will be invoked.

*File: TestAbstraction1.java*

1. **abstract** **class** Shape{
2. **abstract** **void** draw();
3. }
4. //In real scenario, implementation is provided by others i.e. unknown by end user
5. **class** Rectangle **extends** Shape{
6. **void** draw(){System.out.println("drawing rectangle");}
7. }
8. **class** Circle1 **extends** Shape{
9. **void** draw(){System.out.println("drawing circle");}
10. }
11. //In real scenario, method is called by programmer or user
12. **class** TestAbstraction1{
13. **public** **static** **void** main(String args[]){
14. Shape s=**new** Circle1();//In a real scenario, object is provided through method, e.g., getShape() method
15. s.draw();
16. }
17. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestAbstraction1)

drawing circle

Another example of Abstract class in java

*File: TestBank.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. }}

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestBank)

Rate of Interest is: 7 %

Rate of Interest is: 8 %

Abstract class having constructor, data member and methods

An abstract class can have a data member, abstract method, method body (non-abstract method), constructor, and even main() method.

*File: TestAbstraction2.java*

1. //Example of an abstract class that has abstract and non-abstract methods
2. **abstract** **class** Bike{
3. Bike(){System.out.println("bike is created");}
4. **abstract** **void** run();
5. **void** changeGear(){System.out.println("gear changed");}
6. }
7. //Creating a Child class which inherits Abstract class
8. **class** Honda **extends** Bike{
9. **void** run(){System.out.println("running safely..");}
10. }
11. //Creating a Test class which calls abstract and non-abstract methods
12. **class** TestAbstraction2{
13. **public** **static** **void** main(String args[]){
14. Bike obj = **new** Honda();
15. obj.run();
16. obj.changeGear();
17. }
18. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestAbstraction2)

bike is created

running safely..

gear changed

Rule: If there is an abstract method in a class, that class must be abstract.

1. **class** Bike12{
2. **abstract** **void** run();
3. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=Bike12)

compile time error

Rule: If you are extending an abstract class that has an abstract method, you must either provide the implementation of the method or make this class abstract.

Another real scenario of abstract class

The abstract class can also be used to provide some implementation of the [interface](https://www.javatpoint.com/interface-in-java). In such case, the end user may not be forced to override all the methods of the interface.

*Note: If you are beginner to java, learn interface first and skip this example.*

1. **interface** A{
2. **void** a();
3. **void** b();
4. **void** c();
5. **void** d();
6. }
8. **abstract** **class** B **implements** A{
9. **public** **void** c(){System.out.println("I am c");}
10. }
12. **class** M **extends** B{
13. **public** **void** a(){System.out.println("I am a");}
14. **public** **void** b(){System.out.println("I am b");}
15. **public** **void** d(){System.out.println("I am d");}
16. }
18. **class** Test5{
19. **public** **static** **void** main(String args[]){
20. A a=**new** M();
21. a.a();
22. a.b();
23. a.c();
24. a.d();
25. }}

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=Test5)

Output:I am a

I am b

I am c

I am d

# *Interface in Java*

|  |
| --- |
| ***===================================shiv nath sir=============================***  Interfaces  ===========  It says "what to do".  It does not tells "how to do".  Just like class interface will also have data and methods.  but here all data is public, final and static.  all the methods are public and abstract. (However in java 8 and java 9 some new features added which allows some concrete method)  It becomes mandatory for the class which is implementing interface, to implement all the methods of the interface otherwise implementing class will be treated as an abstract class.  We can not create any object of interface. because it has an abstract method and hence abstract method object is not created  Every method which we implement from interface inside class , must be public.  Interface has no constructor.  package mypack;  interface MyInterface-A  {  int MIN = 10; //public, static, final write or not it will be public static final  //public static final int MIN = 10; //write or dont write it will be pub stat final  int MAX = 100;  void fun(); //public, abstract no body or implementation  //public abstract void fun(); wrie or dont write public abstract  void anotherFun(); //Here not necessary to use abstract key word before void fun write or dont write it is an abstract  }  class MyTest implements MyInterface //MyTest is an implementing class  {  //If class MyTest does not implements all the methods of MyInterface then this class  //MyTest will become an abstract class.  public void fun()  {  System.out.println("fun of MyTest");  }  public void anotherFun()  {  System.out.println("anotherFun of MyTest");  }  void diffFun()  {  System.out.println("diffFun of MyTest");  }  }  class InterfaceDemo  {  public static void main(String args[])  {  //new MyInterface() ; //Wrong, we can not create any object or instantiate any interface  MyInterface mif; //-->Has we can not create object but we can create reference vaiable of abstract class i.e reference of child class is stored  mif = new MyTest();  mif.fun(); //OK //It will check in child then in parenṭ  mif.anotherFun(); //OK  //mif.diffFun(); //Error  }  }  ================================================  Creating of an Referece of a child class  ================================================  First is a concrete class or an abstract class - concrete means non abstract class  class Second extends First  {  }  First f = new Second(); //OK  ================================================  First is an interface  class Second implements First  {  }  First f = new Second(); //OK  ================================================  Need of an Interface  ================================================  interface DBDriver  {  //Means standard method is created but it can use in other class if ore data base //increase they will use this functions  //It will give instruction to write the code and connect with the database  void openConnection();  void getConnection();  void closeConnection();  }  class OracleDriver implements DBDriver  {  void openConnection()  {  System.out.println("openConnection of OracleDriver");  }  void getConnection()  {  System.out.println("getConnection of OracleDriver");  }  void closeConnection()  {  System.out.println("closeConnection of OracleDriver");  }  }  class MySqlDriver implements DBDriver  {  void openConnection()  {  System.out.println("openConnection of MySqlDriver");  }  void getConnection()  {  System.out.println("getConnection of MySqlDriver");  }  void closeConnection()  {  System.out.println("closeConnection of MySqlDriver");  }  }  class PostgresDriver implements DBDriver  {  void openConnection()  {  System.out.println("openConnection of PostgresDriver");  }  void getConnection()  {  System.out.println("getConnection of PostgresDriver");  }  void closeConnection()  {  System.out.println("closeConnection of PostgresDriver");  }  }  class DriverDemo  {  public static void main(String args[])  {  DBDriver db; //Interface Name  db = new PostgresDriver(); //Create object for the Implementation  db.openConnection(); //Example of one Interface Multiple Implementation  db = new OracleDriver();  db.openConnection(); //Example of one Interface Multiple Implementation    //db.openConnection(); , db.openConnection(); --->Examople of polymorphism  //One Interface Multiple Implementation  //Bank example -- Central organization atm rule -- atm functionalities  //Implementation varies from class to class so keep it in interface and accordingly implement in each class  //All methods are abstract in interface  //whereas in abstract method may be or may not be abstract method i.e partially abstract  // method is public in interface because it is used by public i.e implemntation by classes so it is public  }  }  Polymorphism: One interface multiple implementation.  Can we implement more than one interface?  YES  We can not inherit more than one classes but we can implement more than one interfac.  =====================================  //Interface Example  interface A  {  void fun();  }  interface B  {  void anotherFun();  }  class C implements A,B //OK  {  void fun(){} //Implemented from interface a  void anotherFun(){} //Implemented from interface b  }  ====================================  We can not inherit more than one classes but we can implement more than one interface.  //Inheritance Example  class A  {  }  class B  {  }  class C extends A,B //ERROR in Inheritance but it is k in interface  {  }  =========================================  class A  {  }  interface B  {  void fun();  }  interface C  {  }  class d  {  }  //An interface can inherit another interface.  interface E extends B // one interface another interface can be done but we have to use extendss not use implements  //if any function is having interface b similar function will be available in interface E with its own functon also  //similarly to child parent relationship it will work  {  void anotherFun();  }  ===============================================  class D implements B,C { ... } //OK  class D extends A { ..... } //OK  class D extends A implements B { ..... } //OK //1st extend then implement so no error  class D implements B extends A { ..... } //Error //implement first extend so error  //Always extends first then implements if we change order their will be an error  class D extends A implements B,C { ..... } //OK  //An interface can inherit another interface.  interface E extends B  // one interface another interface can be done but we have to use extendss not use implements  //if any function is having interface b similar function will be available in interface E with its own functon also  //similarly to child parent relationship it will work  {  void anotherFun();  }  class F implements E { ...... } //Will have to give implementation for fun() as well as anotherFun()  // so automatically it has to call function from interface b and interface E  interface G extends B,C //Solves the issue of multiple inheritance one child has two parents actually it is not mul inhertbut it executes in that way and solves the issue  ===============================================  Relatonship Struture  extends - child parent relationship inherit  implements - implementing all the methods from the interface  1)class extends class  2)interface extends interface  3)class implements interface  [Batsman class] |

An **interface in Java** is a blueprint of a class. It has static constants and abstract methods.

The interface in Java is a mechanism to achieve [*abstraction*](https://www.javatpoint.com/abstract-class-in-java). There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple [inheritance in Java](https://www.javatpoint.com/inheritance-in-java).

In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.

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

It cannot be instantiated just like the abstract class.

Since Java 8, we can have **default and static methods** in an interface.

Since Java 9, we can have **private methods** in an interface.

Why use Java 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.



How to declare an interface?

An interface is declared by using the interface keyword. It provides total abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default. A class that implements an interface must implement all the methods declared in the interface.

Syntax:

1. **interface** <interface\_name>{
3. // declare constant fields
4. // declare methods that abstract
5. // by default.
6. }

Java 8 Interface Improvement

Since [Java 8](https://www.javatpoint.com/java-8-features), interface can have default and static methods which is discussed later.

Internal addition by the compiler

The Java compiler adds public and abstract keywords before the interface method. Moreover, it adds public, static and final keywords before data members.

In other words, Interface fields are public, static and final by default, and the methods are public and abstract.



The relationship between classes and interfaces

As shown in the figure given below, a class extends another class, an interface extends another interface, but a **class implements an interface**.



Java Interface Example

In this example, the Printable interface has only one method, and its implementation is provided in the A6 class.

1. **interface** printable{
2. **void** print();
3. }
4. **class** A6 **implements** printable{
5. **public** **void** print(){System.out.println("Hello");}
7. **public** **static** **void** main(String args[]){
8. A6 obj = **new** A6();
9. obj.print();
10. }
11. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=A6)

Output:

Hello

Java Interface Example: Drawable

In this example, the Drawable interface has only one method. Its implementation is provided by Rectangle and Circle classes. In a real scenario, an interface is defined by someone else, but its implementation is provided by different implementation providers. Moreover, it is used by someone else. The implementation part is hidden by the user who uses the interface.

*File: TestInterface1.java*

1. //Interface declaration: by first user
2. **interface** Drawable{
3. **void** draw();
4. }
5. //Implementation: by second user
6. **class** Rectangle **implements** Drawable{
7. **public** **void** draw(){System.out.println("drawing rectangle");}
8. }
9. **class** Circle **implements** Drawable{
10. **public** **void** draw(){System.out.println("drawing circle");}
11. }
12. //Using interface: by third user
13. **class** TestInterface1{
14. **public** **static** **void** main(String args[]){
15. Drawable d=**new** Circle();//In real scenario, object is provided by method e.g. getDrawable()
16. d.draw();
17. }}

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestInterface1)

Output:

drawing circle

Java Interface Example: Bank

Let's see another example of java interface which provides the implementation of Bank interface.

*File: TestInterface2.java*

1. **interface** Bank{
2. **float** rateOfInterest();
3. }
4. **class** SBI **implements** Bank{
5. **public** **float** rateOfInterest(){**return** 9.15f;}
6. }
7. **class** PNB **implements** Bank{
8. **public** **float** rateOfInterest(){**return** 9.7f;}
9. }
10. **class** TestInterface2{
11. **public** **static** **void** main(String[] args){
12. Bank b=**new** SBI();
13. System.out.println("ROI: "+b.rateOfInterest());
14. }}

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestInterface2)

Output:

ROI: 9.15

Multiple inheritance in Java by interface

If a class implements multiple interfaces, or an interface extends multiple interfaces, it is known as multiple inheritance.



1. **interface** Printable{
2. **void** print();
3. }
4. **interface** Showable{
5. **void** show();
6. }
7. **class** A7 **implements** Printable,Showable{
8. **public** **void** print(){System.out.println("Hello");}
9. **public** **void** show(){System.out.println("Welcome");}
11. **public** **static** **void** main(String args[]){
12. A7 obj = **new** A7();
13. obj.print();
14. obj.show();
15. }
16. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=A7)

Output:Hello

Welcome

Q) Multiple inheritance is not supported through class in java, but it is possible by an interface, why?

As we have explained in the inheritance chapter, multiple inheritance is not supported in the case of [class](https://www.javatpoint.com/object-and-class-in-java) because of ambiguity. However, it is supported in case of an interface because there is no ambiguity. It is because its implementation is provided by the implementation class. For example:

1. **interface** Printable{
2. **void** print();
3. }
4. **interface** Showable{
5. **void** print();
6. }
8. **class** TestInterface3 **implements** Printable, Showable{
9. **public** **void** print(){System.out.println("Hello");}
10. **public** **static** **void** main(String args[]){
11. TestInterface3 obj = **new** TestInterface3();
12. obj.print();
13. }
14. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestInterface3)

Output:

Hello

As you can see in the above example, Printable and Showable interface have same methods but its implementation is provided by class TestTnterface1, so there is no ambiguity.

Interface inheritance

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

1. **interface** Printable{
2. **void** print();
3. }
4. **interface** Showable **extends** Printable{
5. **void** show();
6. }
7. **class** TestInterface4 **implements** Showable{
8. **public** **void** print(){System.out.println("Hello");}
9. **public** **void** show(){System.out.println("Welcome");}
11. **public** **static** **void** main(String args[]){
12. TestInterface4 obj = **new** TestInterface4();
13. obj.print();
14. obj.show();
15. }
16. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestInterface4)

Output:

Hello

Welcome

Java 8 Default Method in Interface

Since Java 8, we can have method body in interface. But we need to make it default method. Let's see an example:

*File: TestInterfaceDefault.java*

1. **interface** Drawable{
2. **void** draw();
3. **default** **void** msg(){System.out.println("default method");}
4. }
5. **class** Rectangle **implements** Drawable{
6. **public** **void** draw(){System.out.println("drawing rectangle");}
7. }
8. **class** TestInterfaceDefault{
9. **public** **static** **void** main(String args[]){
10. Drawable d=**new** Rectangle();
11. d.draw();
12. d.msg();
13. }}

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestInterfaceDefault)

Output:

drawing rectangle

default method

Java 8 Static Method in Interface

Since Java 8, we can have static method in interface. Let's see an example:

*File: TestInterfaceStatic.java*

1. **interface** Drawable{
2. **void** draw();
3. **static** **int** cube(**int** x){**return** x\*x\*x;}
4. }
5. **class** Rectangle **implements** Drawable{
6. **public** **void** draw(){System.out.println("drawing rectangle");}
7. }
9. **class** TestInterfaceStatic{
10. **public** **static** **void** main(String args[]){
11. Drawable d=**new** Rectangle();
12. d.draw();
13. System.out.println(Drawable.cube(3));
14. }}

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestInterfaceStatic)

Output:

drawing rectangle

27

Q) What is marker or tagged interface?

An interface which has no member is known as a marker or tagged interface, for example, [Serializable](https://www.javatpoint.com/serialization-in-java), Cloneable, Remote, etc. They are used to provide some essential information to the JVM so that JVM may perform some useful operation.

1. //How Serializable interface is written?
2. **public** **interface** Serializable{
3. }

Nested Interface in Java

Note: An interface can have another interface which is known as a nested interface. We will learn it in detail in the [nested classes](https://www.javatpoint.com/java-inner-class) chapter. For example:

1. **interface** printable{
2. **void** print();
3. **interface** MessagePrintable{
4. **void** msg();
5. }
6. }

[More about Nested Interface](https://www.javatpoint.com/nested-interface)

Difference between abstract class and interface

Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated.

But there are many differences between abstract class and interface that are given below.

|  |  |
| --- | --- |
| **Abstract class** | **Interface** |
| 1) Abstract class can **have abstract and non-abstract** methods. | Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also. |
| 2) Abstract class **doesn't support multiple inheritance**. | Interface **supports multiple inheritance**. |
| 3) Abstract class **can have final, non-final, static and non-static variables**. | Interface has **only static and final variables**. |
| 4) Abstract class **can provide the implementation of interface**. | Interface **can't provide the implementation of abstract class**. |
| 5) The **abstract keyword** is used to declare abstract class. | The **interface keyword** is used to declare interface. |
| 6) An **abstract class** can extend another Java class and implement multiple Java interfaces. | An **interface** can extend another Java interface only. |
| 7) An **abstract class** can be extended using keyword "extends". | An **interface** can be implemented using keyword "implements". |
| 8) A Java **abstract class** can have class members like private, protected, etc. | Members of a Java interface are public by default. |
| 9)**Example:** public abstract class Shape{ public abstract void draw(); } | **Example:** public interface Drawable{ void draw(); } |

Simply, abstract class achieves partial abstraction (0 to 100%) whereas interface achieves fully abstraction (100%).

Example of abstract class and interface in Java

Let's see a simple example where we are using interface and abstract class both.

1. //Creating interface that has 4 methods
2. **interface** A{
3. **void** a();//bydefault, public and abstract
4. **void** b();
5. **void** c();
6. **void** d();
7. }
9. //Creating abstract class that provides the implementation of one method of A interface
10. **abstract** **class** B **implements** A{
11. **public** **void** c(){System.out.println("I am C");}
12. }
14. //Creating subclass of abstract class, now we need to provide the implementation of rest of the methods
15. **class** M **extends** B{
16. **public** **void** a(){System.out.println("I am a");}
17. **public** **void** b(){System.out.println("I am b");}
18. **public** **void** d(){System.out.println("I am d");}
19. }
21. //Creating a test class that calls the methods of A interface
22. **class** Test5{
23. **public** **static** **void** main(String args[]){
24. A a=**new** M();
25. a.a();
26. a.b();
27. a.c();
28. a.d();
29. }}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Test5)

Output:

I am a

I am b

I am c

I am d

***Java Encapsulation***

1.Package

2.Access Modifiers

3.Encapsulation

**Java Package**

|  |
| --- |
| ***================================shiv nath sir===============================***  packages::::  1.A package is a collection of related classes.  ex java.lang,java.util,java.io,java.sql,etc  package is also simply called directory  (main dir-->sub dir)  java-->util-->Scanner.class==package  import java .lang.\*; //it is by default by java & we do not need to import it explicitly.  -=------------------------------------------  creating our own packages & using it=-=-=-  --------------------------------------------------  create a package(directory) my pack.  Inside mypack ,we will put 2classes - First & Second  We will use this packages 'mypack' in our program.  create dir ---mkdir mypack  create subdir--mkdir src  cd src //create 2 dir - 1- src2-mypack  mkdir mypack  ================================  [First.java]  package mypack;  public class First  {  public void fun()  {  System.out.println("Fun of First");  }  }  ---------------------------------------------------------------  //we have created another file ;  [Second.java]  package mypack;  public class Second  {  public void anotherFun()  {  System.out.println("anotherFun of Second");  }  }  ------------------------------------------------------------------  now we have to use that package so we will go back at src directory so we create file -[UseFirstSecond.java]  import mypack.\*;  class UseFirstSecond  {  public static void main(String args[])  {  First f = new First();  Second s = new Second();  f.fun();  s.anotherFun();  }  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  src (to seperate our package/pgm)  | |--------First.java  |---------->mypack------>  | |-------Second.java  |  |  |-----------UseFirstSecond.java  =====================================  we are in src folder now we have .java file and we also want .class file so we compile the code  To compile::  javac mypack/First.java  javac mypack/Second.java  To use package::  javac UseFirstSecond.java  java UseFirstSecond.java  o/p=  fun of First  anotherFun of Second  Using jar  ---------------------------------------------  to create jar file  jar cf mypack.jar mypack  javac -cp<path to mypack.jar> :.UseFirstSecond.java //compile  java -cp<path to mypack.jar> :.UseFirstSecond.java//run  :. curent folder ke andar ke file ko bhi consider karo  o/p=  fun of First  anotherFun of Second |

A **java package** is a group of similar types of classes, interfaces and sub-packages.

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, javax, swing, net, io, util, sql etc.

Here, we will have the detailed learning of creating and using user-defined packages.

## 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) Java package removes naming collision.



## Simple example of java package

The **package keyword** is used to create a package in java.

1. //save as Simple.java
2. **package** mypack;
3. **public** **class** Simple{
4. **public** **static** **void** main(String args[]){
5. System.out.println("Welcome to package");
6. }
7. }

## How to compile java package

If you are not using any IDE, you need to follow the **syntax** given below:

1. javac -d directory javafilename

For **example**

1. javac -d . Simple.java

The -d switch specifies the destination where to put the generated class file. You can use any directory name like /home (in case of Linux), d:/abc (in case of windows) etc. If you want to keep the package within the same directory, you can use . (dot).

## How to run java package program

You need to use fully qualified name e.g. mypack.Simple etc to run the class.

|  |
| --- |
| **To Compile:** javac -d . Simple.java |
| **To Run:** java mypack.Simple |

Output:Welcome to package

|  |
| --- |
| The -d is a switch that tells the compiler where to put the class file i.e. it represents destination. The . represents the current folder. |

## How to access package from another package?

There are three ways to access the package from outside the package.

1. import package.\*;
2. import package.classname;
3. fully qualified name.

#### 1) Using packagename.\*

If you use package.\* then all the classes and interfaces of this package will be accessible but not subpackages.

The import keyword is used to make the classes and interface of another package accessible to the current package.

## Example of package that import the packagename.\*

1. //save by A.java
2. **package** pack;
3. **public** **class** A{
4. **public** **void** msg(){System.out.println("Hello");}
5. }
6. //save by B.java
7. **package** mypack;
8. **import** pack.\*;
10. **class** B{
11. **public** **static** **void** main(String args[]){
12. A obj = **new** A();
13. obj.msg();
14. }
15. }

Output:Hello

#### 2) Using packagename.classname

If you import package.classname then only declared class of this package will be accessible.

## Example of package by import package.classname

1. //save by A.java
3. **package** pack;
4. **public** **class** A{
5. **public** **void** msg(){System.out.println("Hello");}
6. }
7. //save by B.java
8. **package** mypack;
9. **import** pack.A;
11. **class** B{
12. **public** **static** **void** main(String args[]){
13. A obj = **new** A();
14. obj.msg();
15. }
16. }

Output:Hello

#### 3) Using fully qualified name

If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.

It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

## Example of package by import fully qualified name

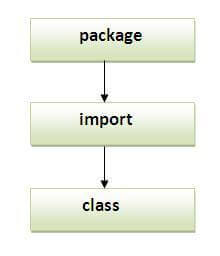
1. //save by A.java
2. **package** pack;
3. **public** **class** A{
4. **public** **void** msg(){System.out.println("Hello");}
5. }
6. //save by B.java
7. **package** mypack;
8. **class** B{
9. **public** **static** **void** main(String args[]){
10. pack.A obj = **new** pack.A();//using fully qualified name
11. obj.msg();
12. }
13. }

Output:Hello

#### Note: If you import a package, subpackages will not be imported.

If you import a package, all the classes and interface of that package will be imported excluding the classes and interfaces of the subpackages. Hence, you need to import the subpackage as well.

#### Note: Sequence of the program must be package then import then class.



## Subpackage in java

Package inside the package is called the **subpackage**. It should be created **to categorize the package further**.

Let's take an example, Sun Microsystem has definded a package named java that contains many classes like System, String, Reader, Writer, Socket etc. These classes represent a particular group e.g. Reader and Writer classes are for Input/Output operation, Socket and ServerSocket classes are for networking etc and so on. So, Sun has subcategorized the java package into subpackages such as lang, net, io etc. and put the Input/Output related classes in io package, Server and ServerSocket classes in net packages and so on.

#### The standard of defining package is domain.company.package e.g. com.javatpoint.bean or org.sssit.dao.

### Example of Subpackage

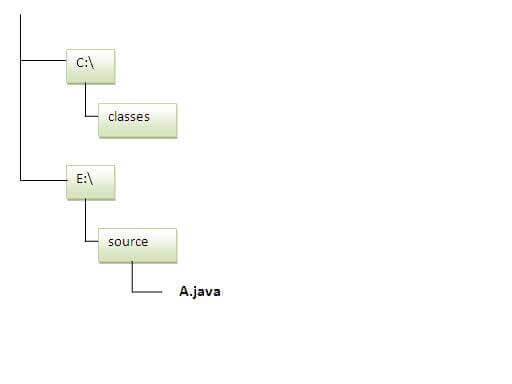
1. **package** com.javatpoint.core;
2. **class** Simple{
3. **public** **static** **void** main(String args[]){
4. System.out.println("Hello subpackage");
5. }
6. }

|  |
| --- |
| **To Compile:** javac -d . Simple.java |
| **To Run:** java com.javatpoint.core.Simple |

Output:Hello subpackage

## How to send the class file to another directory or drive?

There is a scenario, I want to put the class file of A.java source file in classes folder of c: drive. For example:



1. //save as Simple.java
2. **package** mypack;
3. **public** **class** Simple{
4. **public** **static** **void** main(String args[]){
5. System.out.println("Welcome to package");
6. }
7. }

### To Compile:

**e:\sources> javac -d c:\classes Simple.java**

### To Run:

|  |
| --- |
| To run this program from e:\source directory, you need to set classpath of the directory where the class file resides. |
| **e:\sources> set classpath=c:\classes;.;** |
| **e:\sources> java mypack.Simple** |

### Another way to run this program by -classpath switch of java:

The -classpath switch can be used with javac and java tool.

To run this program from e:\source directory, you can use -classpath switch of java that tells where to look for class file. For example:

**e:\sources> java -classpath c:\classes mypack.Simple**

Output:Welcome to package

### Ways to load the class files or jar files

|  |
| --- |
| There are two ways to load the class files temporary and permanent. |

* Temporary
  + By setting the classpath in the command prompt
  + By -classpath switch
* Permanent
  + By setting the classpath in the environment variables
  + By creating the jar file, that contains all the class files, and copying the jar file in the jre/lib/ext folder.

#### Rule: There can be only one public class in a java source file and it must be saved by the public class name.

1. //save as C.java otherwise Compilte Time Error
3. **class** A{}
4. **class** B{}
5. **public** **class** C{}

### How to put two public classes in a package?

|  |
| --- |
| If you want to put two public classes in a package, have two java source files containing one public class, but keep the package name same. For example: |

1. //save as A.java
3. **package** javatpoint;
4. **public** **class** A{}
5. //save as B.java
7. **package** javatpoint;
8. **public** **class** B{}

# Access Modifiers in Java

|  |
| --- |
| ***==============================shiv nath sir===================================***  /\*Access Modifiers or Access Specifiers  Class = Data + Methods  static Data - can access directly using name only no need to create object  non sratic Data - cannot use directly using name only we have to creatre object  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  Keywords used in the Access Specifiers are  1) Public  2) Private  3) Protected  4) Default - You will never write or use Explicity  Example:  Public bike - Anyone can use,chances of bike will get damage  Private bike - only i can use  Protected bike - me and my children can use i.e used in that can access inside the class  but from outside the package it will e accessible within its child class only  Default bike - only access to that class other then class it canno be used  class will define which should be access specifier used i.e public , private,protected or Default  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Object\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  .object - object creation and .object It is used for calling data from one class to other class not fused for calling own class data  Any value data or method we have to access outside the class it is compulsory to create an object and with the help of it we can call it  object can be created anywhere in any calsss where t is required  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  mypack.First package - public int a;  private int b;  protected int c;  int d;  mypack.Second Package  anotherpack.Third Package  mypack.AcessDemo class in mypack paackage - want to print public int a and private int b data we use object  public int a :- print a will work in all of the package and class as it is public / Anyone accessible to anywhere  private int b :- it will be accessible to only mypack.First i.e it own class i cannot be used any other class or any other package /it is accessible to its own class  default int d :- it will be acessible to mypack mypack.First package,mypack.Second Package,mypack.AcessDemo i.e it is used in its own packagee i cannot be used in anotherpackage.Third  mypack.AcessDemo it cannot be used in another  protected :- it is accessible inside the package, outside the class only class which is child class of parent class that can accesss it  Parent classs and its own child class can access the Data  it can access from another package but condition is it shoud be child class of that parent class  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  This rule is applicable for all data and functions  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Specifiers in class\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  similarly for class also we can use specifier  class has two specifiers (exept inner class)  1) Public  2) default  public class :- it means hat class is accessible from anywhere  default class :- it means it can access within the package  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  class First  {  public int a;  private int b; //b can only access in the Class not outside the class  protected int c;  int d;  public void fun1()  {  //Code  //new First();  sop(b);  }  void fun2()  {  //Code  sop(b);  }  }  class Second  {  void fun3(}  {  First first = new First();  sop(first.a);  sop(first.b);  sop(first.c);  sop(first.d);  }  }  //AccessDemo.Java  class AcessDemo  {  public static void main(String args[])  {  First first = new First();  sop(first.b);    Four four = new Four();  sop(four. );  }  package anotherpack;  class Third  {  void fun4(}  {  First first = new First();  sop(first.a);  sop(first.b);  sop(first.c);  sop(first.d);  }  }  //child of First Parent Class  class Four extends First  {  //From parent - a,b,c,d  void fun5(}  {  Four four = new Four();  sop(a); //here we not write .obj because it parent has data so it is part of class or its own family  sop(b); //error  sop(c);  sop(d); //error  }  }  \*/  2 example  /\*  package AccessSpecifiers;  class First  {  public int a;  private int b; //b can only access in the Class not outside the class  protected int c;  int d;  public void fun1()  {  //Code  //new First();  System.out.println(b);  }  void fun2()  {  //Code  System.out.println(b);  }  }  class Second  {  void fun3()  {  First first = new First();  System.out.println(first.a);  System.out.println(first.b); //error  System.out.println(first.c);  System.out.println(first.d);  }  }  //AccessDemo.Java  class AccessDemo  {  public static void main(String args[])  {  First first = new First();  System.out.println(first.a);  //System.out.println(first.b); //error  System.out.println(first.c);  System.out.println(first.d);  //System.out.println(first.b); //error    //Four four = new Four();  //System.out.println(four.d );  }  }  /\*  //package anotherpack;  class Third  {  void fun4(}  {  First first = new First();  System.out.println(first.a);  System.out.println(first.b); //error  System.out.println(first.c);  System.out.println(first.d);  }  }  //child of First Parent Class  class Four extends First  {  //From parent - a,b,c,d  void fun5(}  {  Four four = new Four();  System.out.println(a); //here we not write .obj because it parent has data so it is part of class or its own family  System.out.println(b); //error  System.out.println(c);  System.out.println(d); //error  }  }  \*/ |

There are two types of modifiers in Java: **access modifiers** and **non-access modifiers**.

The access modifiers in Java specifies the accessibility or scope of a field, method, constructor, or class. We can change the access level of fields, constructors, methods, and class by applying the access modifier on it.

There are four types of Java access modifiers:

1. **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
2. **Default**: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
3. **Protected**: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
4. **Public**: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

There are many non-access modifiers, such as static, abstract, synchronized, native, volatile, transient, etc. Here, we are going to learn the access modifiers only.

### Understanding Java Access Modifiers

Let's understand the access modifiers in Java by a simple table.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Access Modifier** | **within class** | **within package** | **outside package by subclass only** | **outside package** |
| **Private** | Y | N | N | N |
| **Default** | Y | Y | N | N |
| **Protected** | Y | Y | Y | N |
| **Public** | Y | Y | Y | Y |

### 1) Private

The private access modifier is accessible only within the class.

**Simple example of private access modifier**

In this example, we have created two classes A and Simple. A class contains private data member and private method. We are accessing these private members from outside the class, so there is a compile-time error.

1. **class** A{
2. **private** **int** data=40;
3. **private** **void** msg(){System.out.println("Hello java");}
4. }
6. **public** **class** Simple{
7. **public** **static** **void** main(String args[]){
8. A obj=**new** A();
9. System.out.println(obj.data);//Compile Time Error
10. obj.msg();//Compile Time Error
11. }
12. }

### Role of Private Constructor

If you make any class constructor private, you cannot create the instance of that class from outside the class. For example:

1. **class** A{
2. **private** A(){}//private constructor
3. **void** msg(){System.out.println("Hello java");}
4. }
5. **public** **class** Simple{
6. **public** **static** **void** main(String args[]){
7. A obj=**new** A();//Compile Time Error
8. }
9. }

#### Note: A class cannot be private or protected except nested class.

### 2) Default

If you don't use any modifier, it is treated as **default** by default. The default modifier is accessible only within package. It cannot be accessed from outside the package. It provides more accessibility than private. But, it is more restrictive than protected, and public.

**Example of default access modifier**

In this example, we have created two packages pack and mypack. We are accessing the A class from outside its package, since A class is not public, so it cannot be accessed from outside the package.

1. //save by A.java
2. **package** pack;
3. **class** A{
4. **void** msg(){System.out.println("Hello");}
5. }
6. //save by B.java
7. **package** mypack;
8. **import** pack.\*;
9. **class** B{
10. **public** **static** **void** main(String args[]){
11. A obj = **new** A();//Compile Time Error
12. obj.msg();//Compile Time Error
13. }
14. }

In the above example, the scope of class A and its method msg() is default so it cannot be accessed from outside the package.

### 3) Protected

The **protected access modifier** is accessible within package and outside the package but through inheritance only.

The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.

It provides more accessibility than the default modifer.

**Example of protected access modifier**

In this example, we have created the two packages pack and mypack. The A class of pack package is public, so can be accessed from outside the package. But msg method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

1. //save by A.java
2. **package** pack;
3. **public** **class** A{
4. **protected** **void** msg(){System.out.println("Hello");}
5. }
6. //save by B.java
7. **package** mypack;
8. **import** pack.\*;
10. **class** B **extends** A{
11. **public** **static** **void** main(String args[]){
12. B obj = **new** B();
13. obj.msg();
14. }
15. }

Output:Hello

### 4) Public

The **public access modifier** is accessible everywhere. It has the widest scope among all other modifiers.

**Example of public access modifier**

1. //save by A.java
3. **package** pack;
4. **public** **class** A{
5. **public** **void** msg(){System.out.println("Hello");}
6. }
7. //save by B.java
9. **package** mypack;
10. **import** pack.\*;
12. **class** B{
13. **public** **static** **void** main(String args[]){
14. A obj = **new** A();
15. obj.msg();
16. }
17. }

Output:Hello

### Java Access Modifiers with Method Overriding

If you are overriding any method, overridden method (i.e. declared in subclass) must not be more restrictive.

1. **class** A{
2. **protected** **void** msg(){System.out.println("Hello java");}
3. }
5. **public** **class** Simple **extends** A{
6. **void** msg(){System.out.println("Hello java");}//C.T.Error
7. **public** **static** **void** main(String args[]){
8. Simple obj=**new** Simple();
9. obj.msg();
10. }
11. }

The default modifier is more restrictive than protected. That is why, there is a compile-time error.

# *Encapsulation in Java*

**Encapsulation in Java** is a process of wrapping code and data together into a single unit, for example, a capsule which is mixed of several medicines.



We can create a fully encapsulated class in Java by making all the data members of the class private. Now we can use setter and getter methods to set and get the data in it.

The **Java Bean** class is the example of a fully encapsulated class.

### Advantage of Encapsulation in Java

By providing only a setter or getter method, you can make the class **read-only or write-only**. In other words, you can skip the getter or setter methods.

It provides you the **control over the data**. Suppose you want to set the value of id which should be greater than 100 only, you can write the logic inside the setter method. You can write the logic not to store the negative numbers in the setter methods.

It is a way to achieve **data hiding** in Java because other class will not be able to access the data through the private data members.

The encapsulate class is **easy to test**. So, it is better for unit testing.

The standard IDE's are providing the facility to generate the getters and setters. So, it is **easy and fast to create an encapsulated class** in Java.

### Simple 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*

1. //A Java class which is a fully encapsulated class.
2. //It has a private data member and getter and setter methods.
3. **package** com.javatpoint;
4. **public** **class** Student{
5. //private data member
6. **private** String name;
7. //getter method for name
8. **public** String getName(){
9. **return** name;
10. }
11. //setter method for name
12. **public** **void** setName(String name){
13. **this**.name=name
14. }
15. }

*File: Test.java*

1. //A Java class to test the encapsulated class.
2. **package** com.javatpoint;
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: java com.javatpoint.Test

Output:

vijay

### Read-Only class

1. //A Java class which has only getter methods.
2. **public** **class** Student{
3. //private data member
4. **private** String college="AKG";
5. //getter method for college
6. **public** String getCollege(){
7. **return** college;
8. }
9. }

Now, you can't change the value of the college data member which is "AKG".

1. s.setCollege("KITE");//will render compile time error

### Write-Only class

1. //A Java class which has only setter methods.
2. **public** **class** Student{
3. //private data member
4. **private** String college;
5. //getter method for college
6. **public** **void** setCollege(String college){
7. **this**.college=college;
8. }
9. }

Now, you can't get the value of the college, you can only change the value of college data member.

1. System.out.println(s.getCollege());//Compile Time Error, because there is no such method
2. System.out.println(s.college);//Compile Time Error, because the college data member is private.
3. //So, it can't be accessed from outside the class

### Another Example of Encapsulation in Java

Let's see another example of encapsulation that has only four fields with its setter and getter methods.

*File: Account.java*

1. //A Account class which is a fully encapsulated class.
2. //It has a private data member and getter and setter methods.
3. **class** Account {
4. //private data members
5. **private** **long** acc\_no;
6. **private** String name,email;
7. **private** **float** amount;
8. //public getter and setter methods
9. **public** **long** getAcc\_no() {
10. **return** acc\_no;
11. }
12. **public** **void** setAcc\_no(**long** acc\_no) {
13. **this**.acc\_no = acc\_no;
14. }
15. **public** String getName() {
16. **return** name;
17. }
18. **public** **void** setName(String name) {
19. **this**.name = name;
20. }
21. **public** String getEmail() {
22. **return** email;
23. }
24. **public** **void** setEmail(String email) {
25. **this**.email = email;
26. }
27. **public** **float** getAmount() {
28. **return** amount;
29. }
30. **public** **void** setAmount(**float** amount) {
31. **this**.amount = amount;
32. }
34. }

*File: TestAccount.java*

1. //A Java class to test the encapsulated class Account.
2. **public** **class** TestEncapsulation {
3. **public** **static** **void** main(String[] args) {
4. //creating instance of Account class
5. Account acc=**new** Account();
6. //setting values through setter methods
7. acc.setAcc\_no(7560504000L);
8. acc.setName("Sonoo Jaiswal");
9. acc.setEmail("sonoojaiswal@javatpoint.com");
10. acc.setAmount(500000f);
11. //getting values through getter methods
12. System.out.println(acc.getAcc\_no()+" "+acc.getName()+" "+acc.getEmail()+" "+acc.getAmount());
13. }
14. }

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=TestEncapsulation)

Output:

7560504000 Sonoo Jaiswal sonoojaiswal@javatpoint.com 500000.0

# *Java Arrays*

Normally, an array is a collection of similar type of elements which has contiguous memory location.

**Java array** is an object which contains elements of a similar data type. Additionally, The elements of an array are stored in a contiguous memory location. It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array.

Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on.

Unlike C/C++, we can get the length of the array using the length member. In C/C++, we need to use the sizeof operator.

In Java, array is an object of a dynamically generated class. Java array inherits the Object class, and implements the Serializable as well as Cloneable interfaces. We can store primitive values or objects in an array in Java. Like C/C++, we can also create single dimentional or multidimentional arrays in Java.

Moreover, Java provides the feature of anonymous arrays which is not available in C/C++.



### Advantages

* **Code Optimization:** It makes the code optimized, we can retrieve or sort the data efficiently.
* **Random access:** We can get any data located at an index position.

### Disadvantages

* **Size Limit:** We can store only the fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in Java which grows automatically.

### Types of Array in java

There are two types of array.

* Single Dimensional Array
* Multidimensional Array

## Single Dimensional Array in Java

**Syntax to Declare an Array in Java**

1. dataType[] arr; (or)
2. dataType []arr; (or)
3. dataType arr[];

**Instantiation of an Array in Java**

1. arrayRefVar=**new** datatype[size];

### Example of Java Array

Let's see the simple example of java array, where we are going to declare, instantiate, initialize and traverse an array.

1. //Java Program to illustrate how to declare, instantiate, initialize
2. //and traverse the Java array.
3. **class** Testarray{
4. **public** **static** **void** main(String args[]){
5. **int** a[]=**new** **int**[5];//declaration and instantiation
6. a[0]=10;//initialization
7. a[1]=20;
8. a[2]=70;
9. a[3]=40;
10. a[4]=50;
11. //traversing array
12. **for**(**int** i=0;i<a.length;i++)//length is the property of array
13. System.out.println(a[i]);
14. }}

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=Testarray)

Output:

10

20

70

40

50

## Declaration, Instantiation and Initialization of Java Array

We can declare, instantiate and initialize the java array together by:

1. **int** a[]={33,3,4,5};//declaration, instantiation and initialization

Let's see the simple example to print this array.

1. //Java Program to illustrate the use of declaration, instantiation
2. //and initialization of Java array in a single line
3. **class** Testarray1{
4. **public** **static** **void** main(String args[]){
5. **int** a[]={33,3,4,5};//declaration, instantiation and initialization
6. //printing array
7. **for**(**int** i=0;i<a.length;i++)//length is the property of array
8. System.out.println(a[i]);
9. }}

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=Testarray1)

Output:

33

3

4

5

## For-each Loop for Java Array

We can also print the Java array using [**for-each loop**](https://www.javatpoint.com/for-each-loop). The Java for-each loop prints the array elements one by one. It holds an array element in a variable, then executes the body of the loop.

The syntax of the for-each loop is given below:

1. **for**(data\_type variable:array){
2. //body of the loop
3. }

Let us see the example of print the elements of Java array using the for-each loop.

1. //Java Program to print the array elements using for-each loop
2. **class** Testarray1{
3. **public** **static** **void** main(String args[]){
4. **int** arr[]={33,3,4,5};
5. //printing array using for-each loop
6. **for**(**int** i:arr)
7. System.out.println(i);
8. }}

Output:

33

3

4

5

## Passing Array to a Method in Java

We can pass the java array to method so that we can reuse the same logic on any array.

Let's see the simple example to get the minimum number of an array using a method.

1. //Java Program to demonstrate the way of passing an array
2. //to method.
3. **class** Testarray2{
4. //creating a method which receives an array as a parameter
5. **static** **void** min(**int** arr[]){
6. **int** min=arr[0];
7. **for**(**int** i=1;i<arr.length;i++)
8. **if**(min>arr[i])
9. min=arr[i];
11. System.out.println(min);
12. }
14. **public** **static** **void** main(String args[]){
15. **int** a[]={33,3,4,5};//declaring and initializing an array
16. min(a);//passing array to method
17. }}

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=Testarray2)

Output:

3

## Anonymous Array in Java

Java supports the feature of an anonymous array, so you don't need to declare the array while passing an array to the method.

1. //Java Program to demonstrate the way of passing an anonymous array
2. //to method.
3. **public** **class** TestAnonymousArray{
4. //creating a method which receives an array as a parameter
5. **static** **void** printArray(**int** arr[]){
6. **for**(**int** i=0;i<arr.length;i++)
7. System.out.println(arr[i]);
8. }
10. **public** **static** **void** main(String args[]){
11. printArray(**new** **int**[]{10,22,44,66});//passing anonymous array to method
12. }}

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestAnonymousArray)

Output:

10

22

44

66

## Returning Array from the Method

We can also return an array from the method in Java.

1. //Java Program to return an array from the method
2. **class** TestReturnArray{
3. //creating method which returns an array
4. **static** **int**[] get(){
5. **return** **new** **int**[]{10,30,50,90,60};
6. }
8. **public** **static** **void** main(String args[]){
9. //calling method which returns an array
10. **int** arr[]=get();
11. //printing the values of an array
12. **for**(**int** i=0;i<arr.length;i++)
13. System.out.println(arr[i]);
14. }}

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestReturnArray)

Output:

10

30

50

90

60

## ArrayIndexOutOfBoundsException

The Java Virtual Machine (JVM) throws an ArrayIndexOutOfBoundsException if length of the array in negative, equal to the array size or greater than the array size while traversing the array.

1. //Java Program to demonstrate the case of
2. //ArrayIndexOutOfBoundsException in a Java Array.
3. **public** **class** TestArrayException{
4. **public** **static** **void** main(String args[]){
5. **int** arr[]={50,60,70,80};
6. **for**(**int** i=0;i<=arr.length;i++){
7. System.out.println(arr[i]);
8. }
9. }}

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestArrayException)

Output:

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4

at TestArrayException.main(TestArrayException.java:5)

50

60

70

80

## Multidimensional Array in Java

In such case, data is stored in row and column based index (also known as matrix form).

**Syntax to Declare Multidimensional Array in Java**

1. dataType[][] arrayRefVar; (or)
2. dataType [][]arrayRefVar; (or)
3. dataType arrayRefVar[][]; (or)
4. dataType []arrayRefVar[];

**Example to instantiate Multidimensional Array in Java**

1. **int**[][] arr=**new** **int**[3][3];//3 row and 3 column

**Example to initialize Multidimensional Array in Java**

1. arr[0][0]=1;
2. arr[0][1]=2;
3. arr[0][2]=3;
4. arr[1][0]=4;
5. arr[1][1]=5;
6. arr[1][2]=6;
7. arr[2][0]=7;
8. arr[2][1]=8;
9. arr[2][2]=9;

### Example of Multidimensional Java Array

Let's see the simple example to declare, instantiate, initialize and print the 2Dimensional array.

1. //Java Program to illustrate the use of multidimensional array
2. **class** Testarray3{
3. **public** **static** **void** main(String args[]){
4. //declaring and initializing 2D array
5. **int** arr[][]={{1,2,3},{2,4,5},{4,4,5}};
6. //printing 2D array
7. **for**(**int** i=0;i<3;i++){
8. **for**(**int** j=0;j<3;j++){
9. System.out.print(arr[i][j]+" ");
10. }
11. System.out.println();
12. }
13. }}

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=Testarray3)

Output:

1 2 3

2 4 5

4 4 5

## Jagged Array in Java

If we are creating odd number of columns in a 2D array, it is known as a jagged array. In other words, it is an array of arrays with different number of columns.

1. //Java Program to illustrate the jagged array
2. **class** TestJaggedArray{
3. **public** **static** **void** main(String[] args){
4. //declaring a 2D array with odd columns
5. **int** arr[][] = **new** **int**[3][];
6. arr[0] = **new** **int**[3];
7. arr[1] = **new** **int**[4];
8. arr[2] = **new** **int**[2];
9. //initializing a jagged array
10. **int** count = 0;
11. **for** (**int** i=0; i<arr.length; i++)
12. **for**(**int** j=0; j<arr[i].length; j++)
13. arr[i][j] = count++;
15. //printing the data of a jagged array
16. **for** (**int** i=0; i<arr.length; i++){
17. **for** (**int** j=0; j<arr[i].length; j++){
18. System.out.print(arr[i][j]+" ");
19. }
20. System.out.println();//new line
21. }
22. }
23. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestJaggedArray)

Output:

0 1 2

3 4 5 6

7 8

## What is the class name of Java array?

In Java, an array is an object. For array object, a proxy class is created whose name can be obtained by getClass().getName() method on the object.

1. //Java Program to get the class name of array in Java
2. **class** Testarray4{
3. **public** **static** **void** main(String args[]){
4. //declaration and initialization of array
5. **int** arr[]={4,4,5};
6. //getting the class name of Java array
7. Class c=arr.getClass();
8. String name=c.getName();
9. //printing the class name of Java array
10. System.out.println(name);
12. }}

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=Testarray4)

Output:

I

## Copying a Java Array

We can copy an array to another by the arraycopy() method of System class.

**Syntax of arraycopy method**

1. **public** **static** **void** arraycopy(
2. Object src, **int** srcPos,Object dest, **int** destPos, **int** length
3. )

### Example of Copying an Array in Java

1. //Java Program to copy a source array into a destination array in Java
2. **class** TestArrayCopyDemo {
3. **public** **static** **void** main(String[] args) {
4. //declaring a source array
5. **char**[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e',
6. 'i', 'n', 'a', 't', 'e', 'd' };
7. //declaring a destination array
8. **char**[] copyTo = **new** **char**[7];
9. //copying array using System.arraycopy() method
10. System.arraycopy(copyFrom, 2, copyTo, 0, 7);
11. //printing the destination array
12. System.out.println(String.valueOf(copyTo));
13. }
14. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestArrayCopyDemo)

Output:

caffein

## Cloning an Array in Java

Since, Java array implements the Cloneable interface, we can create the clone of the Java array. If we create the clone of a single-dimensional array, it creates the deep copy of the Java array. It means, it will copy the actual value. But, if we create the clone of a multidimensional array, it creates the shallow copy of the Java array which means it copies the references.

1. //Java Program to clone the array
2. **class** Testarray1{
3. **public** **static** **void** main(String args[]){
4. **int** arr[]={33,3,4,5};
5. System.out.println("Printing original array:");
6. **for**(**int** i:arr)
7. System.out.println(i);
9. System.out.println("Printing clone of the array:");
10. **int** carr[]=arr.clone();
11. **for**(**int** i:carr)
12. System.out.println(i);
14. System.out.println("Are both equal?");
15. System.out.println(arr==carr);
17. }}

Output:

Printing original array:

33

3

4

5

Printing clone of the array:

33

3

4

5

Are both equal?

false

## Addition of 2 Matrices in Java

Let's see a simple example that adds two matrices.

1. //Java Program to demonstrate the addition of two matrices in Java
2. **class** Testarray5{
3. **public** **static** **void** main(String args[]){
4. //creating two matrices
5. **int** a[][]={{1,3,4},{3,4,5}};
6. **int** b[][]={{1,3,4},{3,4,5}};
8. //creating another matrix to store the sum of two matrices
9. **int** c[][]=**new** **int**[2][3];
11. //adding and printing addition of 2 matrices
12. **for**(**int** i=0;i<2;i++){
13. **for**(**int** j=0;j<3;j++){
14. c[i][j]=a[i][j]+b[i][j];
15. System.out.print(c[i][j]+" ");
16. }
17. System.out.println();//new line
18. }
20. }}

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=Testarray5)

Output:

2 6 8

6 8 10

## Multiplication of 2 Matrices in Java

In the case of matrix multiplication, a one-row element of the first matrix is multiplied by all the columns of the second matrix which can be understood by the image given below.



Let's see a simple example to multiply two matrices of 3 rows and 3 columns.

1. //Java Program to multiply two matrices
2. **public** **class** MatrixMultiplicationExample{
3. **public** **static** **void** main(String args[]){
4. //creating two matrices
5. **int** a[][]={{1,1,1},{2,2,2},{3,3,3}};
6. **int** b[][]={{1,1,1},{2,2,2},{3,3,3}};
8. //creating another matrix to store the multiplication of two matrices
9. **int** c[][]=**new** **int**[3][3];  //3 rows and 3 columns
11. //multiplying and printing multiplication of 2 matrices
12. **for**(**int** i=0;i<3;i++){
13. **for**(**int** j=0;j<3;j++){
14. c[i][j]=0;
15. **for**(**int** k=0;k<3;k++)
16. {
17. c[i][j]+=a[i][k]\*b[k][j];
18. }//end of k loop
19. System.out.print(c[i][j]+" ");  //printing matrix element
20. }//end of j loop
21. System.out.println();//new line
22. }
23. }}

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=MatrixMultiplicationExample)

Output:

6 6 6

12 12 12

18 18 18

# *Java String*

====================================================================================

* In [Java](https://www.javatpoint.com/java-tutorial), string is basically an object that represents sequence of char values. An [array](https://www.javatpoint.com/array-in-java) of characters works same as Java string. For example:

1. **char**[] ch={'j','a','v','a','t','p','o','i','n','t'};
2. String s=**new** String(ch);

is same as:

1. String s="javatpoint";

* **Java String** class provides a lot of methods to perform operations on strings such as compare(), concat(), equals(), split(), length(), replace(), compareTo(), intern(), substring() etc.

The java.lang.String class implements Serializable, Comparable and CharSequence [interfaces](https://www.javatpoint.com/interface-in-java).



## CharSequence Interface

* The CharSequence interface is used to represent the sequence of characters. String, [StringBuffer](https://www.javatpoint.com/StringBuffer-class) and [StringBuilder](https://www.javatpoint.com/StringBuilder-class) classes implement it. It means, we can create strings in java by using these three classes.



* The Java String is immutable which means it cannot be changed. Whenever we change any string, a new instance is created. For mutable strings, you can use StringBuffer and StringBuilder classes.
* We will discuss immutable string later. Let's first understand what is String in Java and how to create the String object.

### What is String in java

* Generally, String is a sequence of characters. But in Java, string is an object that represents a sequence of characters. The java.lang.String class is used to create a string object.

### How to create a string object?

There are two ways to create String object:

1. By string literal
2. By new keyword

### 1) String Literal

* Java String literal is created by using double quotes. For Example:

1. String s="welcome";

* Each time you create a string literal, the JVM checks the "string constant pool" first. If the string already exists in the pool, a reference to the pooled instance is returned. If the string doesn't exist in the pool, a new string instance is created and placed in the pool. For example:

1. String s1="Welcome";
2. String s2="Welcome";//It doesn't create a new instance



* In the above example, only one object will be created. Firstly, JVM will not find any string object with the value "Welcome" in string constant pool, that is why it will create a new object. After that it will find the string with the value "Welcome" in the pool, it will not create a new object but will return the reference to the same instance.

#### Note: String objects are stored in a special memory area known as the "string constant pool".

### Why Java uses the concept of String literal?

To make Java more memory efficient (because no new objects are created if it exists already in the string constant pool).

### 2) By new keyword

1. String s=**new** String("Welcome");//creates two objects and one reference variable

In such case, [JVM](https://www.javatpoint.com/jvm-java-virtual-machine) will create a new string object in normal (non-pool) heap memory, and the literal "Welcome" will be placed in the string constant pool. The variable s will refer to the object in a heap (non-pool).

### Java String Example

1. **public** **class** StringExample{
2. **public** **static** **void** main(String args[]){
3. String s1="java";//creating string by java string literal
4. **char** ch[]={'s','t','r','i','n','g','s'};
5. String s2=**new** String(ch);//converting char array to string
6. String s3=**new** String("example");//creating java string by new keyword
7. System.out.println(s1);
8. System.out.println(s2);
9. System.out.println(s3);
10. }}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=StringExample)

java

strings

example

### Java String class methods

The java.lang.String class provides many useful methods to perform operations on sequence of char values.

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1 | [char charAt(int index)](https://www.javatpoint.com/java-string-charat) | returns char value for the particular index |
| 2 | [int length()](https://www.javatpoint.com/java-string-length) | returns string length |
| 3 | [static String format(String format, Object... args)](https://www.javatpoint.com/java-string-format) | returns a formatted string. |
| 4 | [static String format(Locale l, String format, Object... args)](https://www.javatpoint.com/java-string-format) | returns formatted string with given locale. |
| 5 | [String substring(int beginIndex)](https://www.javatpoint.com/java-string-substring) | returns substring for given begin index. |
| 6 | [String substring(int beginIndex, int endIndex)](https://www.javatpoint.com/java-string-substring) | returns substring for given begin index and end index. |
| 7 | [boolean contains(CharSequence s)](https://www.javatpoint.com/java-string-contains) | returns true or false after matching the sequence of char value. |
| 8 | [static String join(CharSequence delimiter, CharSequence... elements)](https://www.javatpoint.com/java-string-join) | returns a joined string. |
| 9 | [static String join(CharSequence delimiter, Iterable<? extends CharSequence> elements)](https://www.javatpoint.com/java-string-join) | returns a joined string. |
| 10 | [boolean equals(Object another)](https://www.javatpoint.com/java-string-equals) | checks the equality of string with the given object. |
| 11 | [boolean isEmpty()](https://www.javatpoint.com/java-string-isempty) | checks if string is empty. |
| 12 | [String concat(String str)](https://www.javatpoint.com/java-string-concat) | concatenates the specified string. |
| 13 | [String replace(char old, char new)](https://www.javatpoint.com/java-string-replace) | replaces all occurrences of the specified char value. |
| 14 | [String replace(CharSequence old, CharSequence new)](https://www.javatpoint.com/java-string-replace) | replaces all occurrences of the specified CharSequence. |
| 15 | [static String equalsIgnoreCase(String another)](https://www.javatpoint.com/java-string-equalsignorecase) | compares another string. It doesn't check case. |
| 16 | [String[] split(String regex)](https://www.javatpoint.com/java-string-split) | returns a split string matching regex. |
| 17 | [String[] split(String regex, int limit)](https://www.javatpoint.com/java-string-split) | returns a split string matching regex and limit. |
| 18 | [String intern()](https://www.javatpoint.com/java-string-intern) | returns an interned string. |
| 19 | [int indexOf(int ch)](https://www.javatpoint.com/java-string-indexof) | returns the specified char value index. |
| 20 | [int indexOf(int ch, int fromIndex)](https://www.javatpoint.com/java-string-indexof) | returns the specified char value index starting with given index. |
| 21 | [int indexOf(String substring)](https://www.javatpoint.com/java-string-indexof) | returns the specified substring index. |
| 22 | [int indexOf(String substring, int fromIndex)](https://www.javatpoint.com/java-string-indexof) | returns the specified substring index starting with given index. |
| 23 | [String toLowerCase()](https://www.javatpoint.com/java-string-tolowercase) | returns a string in lowercase. |
| 24 | [String toLowerCase(Locale l)](https://www.javatpoint.com/java-string-tolowercase) | returns a string in lowercase using specified locale. |
| 25 | [String toUpperCase()](https://www.javatpoint.com/java-string-touppercase) | returns a string in uppercase. |
| 26 | [String toUpperCase(Locale l)](https://www.javatpoint.com/java-string-touppercase) | returns a string in uppercase using specified locale. |
| 27 | [String trim()](https://www.javatpoint.com/java-string-trim) | removes beginning and ending spaces of this string. |
| 28 | [static String valueOf(int value)](https://www.javatpoint.com/java-string-valueof) | converts given type into string. It is an overloaded method. |

# Immutable String in Java:

* In java, **string objects are immutable**. Immutable simply means unmodifiable or unchangeable.
* Once string object is created its data or state can't be changed but a new string object is created.

Let's try to understand the immutability concept by the example given below:

1. **class** Testimmutablestring{
2. **public** **static** **void** main(String args[]){
3. String s="Sachin";
4. s.concat(" Tendulkar");//concat() method appends the string at the end
5. System.out.println(s);//will print Sachin because strings are immutable objects
6. }
7. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Testimmutablestring)

Output:Sachin

Now it can be understood by the diagram given below. Here Sachin is not changed but a new object is created with sachintendulkar. That is why string is known as immutable.



* As you can see in the above figure that two objects are created but s reference variable still refers to "Sachin" not to "Sachin Tendulkar".
* But if we explicitely assign it to the reference variable, it will refer to "Sachin Tendulkar" object.For example:

1. **class** Testimmutablestring1{
2. **public** **static** **void** main(String args[]){
3. String s="Sachin";
4. s=s.concat(" Tendulkar");
5. System.out.println(s);
6. }
7. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Testimmutablestring1)

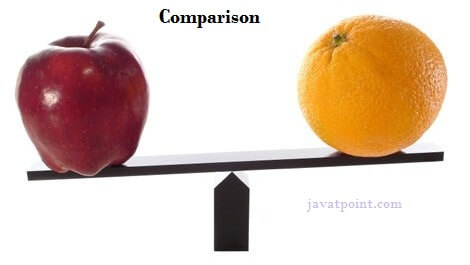
Output:Sachin Tendulkar

In such case, s points to the "Sachin Tendulkar". Please notice that still sachin object is not modified.

### Why string objects are immutable in java?

|  |
| --- |
| * Because java uses the concept of string literal.Suppose there are 5 reference variables,all referes to one object "sachin".If one reference variable changes the value of the object, it will be affected to all the reference variables. That is why string objects are immutable in java. |

# Java String compare



We can compare string in java on the basis of content and reference.

It is used in **authentication** (by equals() method), **sorting** (by compareTo() method), **reference matching** (by == operator) etc.

There are three ways to compare string in java:

1. By equals() method
2. By = = operator
3. By compareTo() method

## 1) String compare by equals() method

The String equals() method compares the original content of the string. It compares values of string for equality. String class provides two methods:

* **public boolean equals(Object another)** compares this string to the specified object.
* **public boolean equalsIgnoreCase(String another)** compares this String to another string, ignoring case.

1. **class** Teststringcomparison1{
2. **public** **static** **void** main(String args[]){
3. String s1="Sachin";
4. String s2="Sachin";
5. String s3=**new** String("Sachin");
6. String s4="Saurav";
7. System.out.println(s1.equals(s2));//true
8. System.out.println(s1.equals(s3));//true
9. System.out.println(s1.equals(s4));//false
10. }
11. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Teststringcomparison1)

Output:true

true

false

1. **class** Teststringcomparison2{
2. **public** **static** **void** main(String args[]){
3. String s1="Sachin";
4. String s2="SACHIN";
6. System.out.println(s1.equals(s2));//false
7. System.out.println(s1.equalsIgnoreCase(s2));//true
8. }
9. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Teststringcomparison2)

Output:

false

true

## 2) String compare by == operator

The = = operator compares references not values.

1. **class** Teststringcomparison3{
2. **public** **static** **void** main(String args[]){
3. String s1="Sachin";
4. String s2="Sachin";
5. String s3=**new** String("Sachin");
6. System.out.println(s1==s2);//true (because both refer to same instance)
7. System.out.println(s1==s3);//false(because s3 refers to instance created in nonpool)
8. }
9. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Teststringcomparison3)

Output:true

false

## 3) String compare by compareTo() method

The String compareTo() method compares values lexicographically and returns an integer value that describes if first string is less than, equal to or greater than second string.

Suppose s1 and s2 are two string variables. If:

* **s1 == s2** :0
* **s1 > s2**  :positive value
* **s1 < s2**  :negative value

1. **class** Teststringcomparison4{
2. **public** **static** **void** main(String args[]){
3. String s1="Sachin";
4. String s2="Sachin";
5. String s3="Ratan";
6. System.out.println(s1.compareTo(s2));//0
7. System.out.println(s1.compareTo(s3));//1(because s1>s3)
8. System.out.println(s3.compareTo(s1));//-1(because s3 < s1 )
9. }
10. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Teststringcomparison4)

Output:0

1

-1

# String Concatenation in Java

In java, string concatenation forms a new string that is the combination of multiple strings. There are two ways to concat string in java:

1. By + (string concatenation) operator
2. By concat() method

## 1) String Concatenation by + (string concatenation) operator

Java string concatenation operator (+) is used to add strings. For Example:

1. **class** TestStringConcatenation1{
2. **public** **static** **void** main(String args[]){
3. String s="Sachin"+" Tendulkar";
4. System.out.println(s);//Sachin Tendulkar
5. }
6. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestStringConcatenation1)

Output:Sachin Tendulkar

The **Java compiler transforms** above code to this:

1. String s=(**new** StringBuilder()).append("Sachin").append(" Tendulkar).toString();

In java, String concatenation is implemented through the StringBuilder (or StringBuffer) class and its append method. String concatenation operator produces a new string by appending the second operand onto the end of the first operand. The string concatenation operator can concat not only string but primitive values also. For Example:

1. **class** TestStringConcatenation2{
2. **public** **static** **void** main(String args[]){
3. String s=50+30+"Sachin"+40+40;
4. System.out.println(s);//80Sachin4040
5. }
6. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestStringConcatenation2)

80Sachin4040

#### Note: After a string literal, all the + will be treated as string concatenation operator.

### 2) String Concatenation by concat() method

The String concat() method concatenates the specified string to the end of current string. Syntax:

1. **public** String concat(String another)

Let's see the example of String concat() method.

1. **class** TestStringConcatenation3{
2. **public** **static** **void** main(String args[]){
3. String s1="Sachin ";
4. String s2="Tendulkar";
5. String s3=s1.concat(s2);
6. System.out.println(s3);//Sachin Tendulkar
7. }
8. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestStringConcatenation3)

Sachin Tendulkar

# *Substring in Java*

A part of string is called **substring**. In other words, substring is a subset of another string. In case of substring startIndex is inclusive and endIndex is exclusive.

#### Note: Index starts from 0.

You can get substring from the given string object by one of the two methods:

1. **public String substring(int startIndex):** This method returns new String object containing the substring of the given string from specified startIndex (inclusive).
2. **public String substring(int startIndex, int endIndex):**This method returns new String object containing the substring of the given string from specified startIndex to endIndex.

In case of string:

* **startIndex:** inclusive
* **endIndex:** exclusive

Let's understand the startIndex and endIndex by the code given below.

1. String s="hello";
2. System.out.println(s.substring(0,2));//he

In the above substring, 0 points to h but 2 points to e (because end index is exclusive).

## Example of java substring

1. **public** **class** TestSubstring{
2. **public** **static** **void** main(String args[]){
3. String s="SachinTendulkar";
4. System.out.println(s.substring(6));//Tendulkar
5. System.out.println(s.substring(0,6));//Sachin
6. }
7. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestSubstring)

Tendulkar

Sachin

# Java String class methods

The java.lang.String class provides a lot of methods to work on string. By the help of these methods, we can perform operations on string such as trimming, concatenating, converting, comparing, replacing strings etc.

Java String is a powerful concept because everything is treated as a string if you submit any form in window based, web based or mobile application.

Let's see the important methods of String class.

### Java String toUpperCase() and toLowerCase() method

The java string toUpperCase() method converts this string into uppercase letter and string toLowerCase() method into lowercase letter.

1. String s="Sachin";
2. System.out.println(s.toUpperCase());//SACHIN
3. System.out.println(s.toLowerCase());//sachin
4. System.out.println(s);//Sachin(no change in original)

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Testmethodofstringclass)

SACHIN

sachin

Sachin

### Java String trim() method

The string trim() method eliminates white spaces before and after string.

1. String s="  Sachin  ";
2. System.out.println(s);//  Sachin
3. System.out.println(s.trim());//Sachin

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Testmethodofstringclass1)

Sachin

Sachin

### Java String startsWith() and endsWith() method

1. String s="Sachin";
2. System.out.println(s.startsWith("Sa"));//true
3. System.out.println(s.endsWith("n"));//true

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Testmethodofstringclass2)

true

true

### Java String charAt() method

The string charAt() method returns a character at specified index.

1. String s="Sachin";
2. System.out.println(s.charAt(0));//S
3. System.out.println(s.charAt(3));//h

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Testmethodofstringclass3)

S

h

### Java String length() method

The string length() method returns length of the string.

1. String s="Sachin";
2. System.out.println(s.length());//6

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Testmethodofstringclass4)

6

### Java String intern() method

A pool of strings, initially empty, is maintained privately by the class String.

When the intern method is invoked, if the pool already contains a string equal to this String object as determined by the equals(Object) method, then the string from the pool is returned. Otherwise, this String object is added to the pool and a reference to this String object is returned.

1. String s=**new** String("Sachin");
2. String s2=s.intern();
3. System.out.println(s2);//Sachin

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Testmethodofstringclass5)

Sachin

### Java String valueOf() method

The string valueOf() method coverts given type such as int, long, float, double, boolean, char and char array into string.

1. **int** a=10;
2. String s=String.valueOf(a);
3. System.out.println(s+10);

Output:

1010

### Java String replace() method

The string replace() method replaces all occurrence of first sequence of character with second sequence of character.

1. String s1="Java is a programming language. Java is a platform. Java is an Island.";
2. String replaceString=s1.replace("Java","Kava");//replaces all occurrences of "Java" to "Kava"
3. System.out.println(replaceString);

Output:

Kava is a programming language. Kava is a platform. Kava is an Island.

# Java StringBuffer class

Java StringBuffer class is used to create mutable (modifiable) string. The StringBuffer class in java is same as String class except it is mutable i.e. it can be changed.

#### Note: Java StringBuffer class is thread-safe i.e. multiple threads cannot access it simultaneously. So it is safe and will result in an order.

### Important Constructors of StringBuffer class

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| StringBuffer() | creates an empty string buffer with the initial capacity of 16. |
| StringBuffer(String str) | creates a string buffer with the specified string. |
| StringBuffer(int capacity) | creates an empty string buffer with the specified capacity as length. |

### Important methods of StringBuffer class

|  |  |  |
| --- | --- | --- |
| **Modifier and Type** | **Method** | **Description** |
| public synchronized StringBuffer | append(String s) | is used to append the specified string with this string. The append() method is overloaded like append(char), append(boolean), append(int), append(float), append(double) etc. |
| public synchronized StringBuffer | insert(int offset, String s) | is used to insert the specified string with this string at the specified position. The insert() method is overloaded like insert(int, char), insert(int, boolean), insert(int, int), insert(int, float), insert(int, double) etc. |
| public synchronized StringBuffer | replace(int startIndex, int endIndex, String str) | is used to replace the string from specified startIndex and endIndex. |
| public synchronized StringBuffer | delete(int startIndex, int endIndex) | is used to delete the string from specified startIndex and endIndex. |
| public synchronized StringBuffer | reverse() | is used to reverse the string. |
| public int | capacity() | is used to return the current capacity. |
| public void | ensureCapacity(int minimumCapacity) | is used to ensure the capacity at least equal to the given minimum. |
| public char | charAt(int index) | is used to return the character at the specified position. |
| public int | length() | is used to return the length of the string i.e. total number of characters. |
| public String | substring(int beginIndex) | is used to return the substring from the specified beginIndex. |
| public String | substring(int beginIndex, int endIndex) | is used to return the substring from the specified beginIndex and endIndex. |

### What is mutable string

A string that can be modified or changed is known as mutable string. StringBuffer and StringBuilder classes are used for creating mutable string.

### 1) StringBuffer append() method

The append() method concatenates the given argument with this string.

1. **class** StringBufferExample{
2. **public** **static** **void** main(String args[]){
3. StringBuffer sb=**new** StringBuffer("Hello ");
4. sb.append("Java");//now original string is changed
5. System.out.println(sb);//prints Hello Java
6. }
7. }

### 2) StringBuffer insert() method

The insert() method inserts the given string with this string at the given position.

1. **class** StringBufferExample2{
2. **public** **static** **void** main(String args[]){
3. StringBuffer sb=**new** StringBuffer("Hello ");
4. sb.insert(1,"Java");//now original string is changed
5. System.out.println(sb);//prints HJavaello
6. }
7. }

### 3) StringBuffer replace() method

The replace() method replaces the given string from the specified beginIndex and endIndex.

1. **class** StringBufferExample3{
2. **public** **static** **void** main(String args[]){
3. StringBuffer sb=**new** StringBuffer("Hello");
4. sb.replace(1,3,"Java");
5. System.out.println(sb);//prints HJavalo
6. }
7. }

### 4) StringBuffer delete() method

The delete() method of StringBuffer class deletes the string from the specified beginIndex to endIndex.

1. **class** StringBufferExample4{
2. **public** **static** **void** main(String args[]){
3. StringBuffer sb=**new** StringBuffer("Hello");
4. sb.delete(1,3);
5. System.out.println(sb);//prints Hlo
6. }
7. }

### 5) StringBuffer reverse() method

The reverse() method of StringBuilder class reverses the current string.

1. **class** StringBufferExample5{
2. **public** **static** **void** main(String args[]){
3. StringBuffer sb=**new** StringBuffer("Hello");
4. sb.reverse();
5. System.out.println(sb);//prints olleH
6. }
7. }

### 6) StringBuffer capacity() method

The capacity() method of StringBuffer class returns the current capacity of the buffer. The default capacity of the buffer is 16. If the number of character increases from its current capacity, it increases the capacity by (oldcapacity\*2)+2. For example if your current capacity is 16, it will be (16\*2)+2=34.

1. **class** StringBufferExample6{
2. **public** **static** **void** main(String args[]){
3. StringBuffer sb=**new** StringBuffer();
4. System.out.println(sb.capacity());//default 16
5. sb.append("Hello");
6. System.out.println(sb.capacity());//now 16
7. sb.append("java is my favourite language");
8. System.out.println(sb.capacity());//now (16\*2)+2=34 i.e (oldcapacity\*2)+2
9. }
10. }

### 7) StringBuffer ensureCapacity() method

The ensureCapacity() method of StringBuffer class ensures that the given capacity is the minimum to the current capacity. If it is greater than the current capacity, it increases the capacity by (oldcapacity\*2)+2. For example if your current capacity is 16, it will be (16\*2)+2=34.

1. **class** StringBufferExample7{
2. **public** **static** **void** main(String args[]){
3. StringBuffer sb=**new** StringBuffer();
4. System.out.println(sb.capacity());//default 16
5. sb.append("Hello");
6. System.out.println(sb.capacity());//now 16
7. sb.append("java is my favourite language");
8. System.out.println(sb.capacity());//now (16\*2)+2=34 i.e (oldcapacity\*2)+2
9. sb.ensureCapacity(10);//now no change
10. System.out.println(sb.capacity());//now 34
11. sb.ensureCapacity(50);//now (34\*2)+2
12. System.out.println(sb.capacity());//now 70
13. }
14. }

# Java StringBuilder class

Java StringBuilder class is used to create mutable (modifiable) string. The Java StringBuilder class is same as StringBuffer class except that it is non-synchronized. It is available since JDK 1.5.

## Important Constructors of StringBuilder class

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| StringBuilder() | creates an empty string Builder with the initial capacity of 16. |
| StringBuilder(String str) | creates a string Builder with the specified string. |
| StringBuilder(int length) | creates an empty string Builder with the specified capacity as length. |

## Important methods of StringBuilder class

|  |  |
| --- | --- |
| **Method** | **Description** |
| public StringBuilder append(String s) | is used to append the specified string with this string. The append() method is overloaded like append(char), append(boolean), append(int), append(float), append(double) etc. |
| public StringBuilder insert(int offset, String s) | is used to insert the specified string with this string at the specified position. The insert() method is overloaded like insert(int, char), insert(int, boolean), insert(int, int), insert(int, float), insert(int, double) etc. |
| public StringBuilder replace(int startIndex, int endIndex, String str) | is used to replace the string from specified startIndex and endIndex. |
| public StringBuilder delete(int startIndex, int endIndex) | is used to delete the string from specified startIndex and endIndex. |
| public StringBuilder reverse() | is used to reverse the string. |
| public int capacity() | is used to return the current capacity. |
| public void ensureCapacity(int minimumCapacity) | is used to ensure the capacity at least equal to the given minimum. |
| public char charAt(int index) | is used to return the character at the specified position. |
| public int length() | is used to return the length of the string i.e. total number of characters. |
| public String substring(int beginIndex) | is used to return the substring from the specified beginIndex. |
| public String substring(int beginIndex, int endIndex) | is used to return the substring from the specified beginIndex and endIndex. |

## Java StringBuilder Examples

Let's see the examples of different methods of StringBuilder class.

### 1) StringBuilder append() method

The StringBuilder append() method concatenates the given argument with this string.

1. **class** StringBuilderExample{
2. **public** **static** **void** main(String args[]){
3. StringBuilder sb=**new** StringBuilder("Hello ");
4. sb.append("Java");//now original string is changed
5. System.out.println(sb);//prints Hello Java
6. }
7. }

### 2) StringBuilder insert() method

The StringBuilder insert() method inserts the given string with this string at the given position.

1. **class** StringBuilderExample2{
2. **public** **static** **void** main(String args[]){
3. StringBuilder sb=**new** StringBuilder("Hello ");
4. sb.insert(1,"Java");//now original string is changed
5. System.out.println(sb);//prints HJavaello
6. }
7. }

### 3) StringBuilder replace() method

The StringBuilder replace() method replaces the given string from the specified beginIndex and endIndex.

1. **class** StringBuilderExample3{
2. **public** **static** **void** main(String args[]){
3. StringBuilder sb=**new** StringBuilder("Hello");
4. sb.replace(1,3,"Java");
5. System.out.println(sb);//prints HJavalo
6. }
7. }

### 4) StringBuilder delete() method

The delete() method of StringBuilder class deletes the string from the specified beginIndex to endIndex.

1. **class** StringBuilderExample4{
2. **public** **static** **void** main(String args[]){
3. StringBuilder sb=**new** StringBuilder("Hello");
4. sb.delete(1,3);
5. System.out.println(sb);//prints Hlo
6. }
7. }

### 5) StringBuilder reverse() method

The reverse() method of StringBuilder class reverses the current string.

1. **class** StringBuilderExample5{
2. **public** **static** **void** main(String args[]){
3. StringBuilder sb=**new** StringBuilder("Hello");
4. sb.reverse();
5. System.out.println(sb);//prints olleH
6. }
7. }

### 6) StringBuilder capacity() method

The capacity() method of StringBuilder class returns the current capacity of the Builder. The default capacity of the Builder is 16. If the number of character increases from its current capacity, it increases the capacity by (oldcapacity\*2)+2. For example if your current capacity is 16, it will be (16\*2)+2=34.

1. **class** StringBuilderExample6{
2. **public** **static** **void** main(String args[]){
3. StringBuilder sb=**new** StringBuilder();
4. System.out.println(sb.capacity());//default 16
5. sb.append("Hello");
6. System.out.println(sb.capacity());//now 16
7. sb.append("java is my favourite language");
8. System.out.println(sb.capacity());//now (16\*2)+2=34 i.e (oldcapacity\*2)+2
9. }
10. }

### 7) StringBuilder ensureCapacity() method

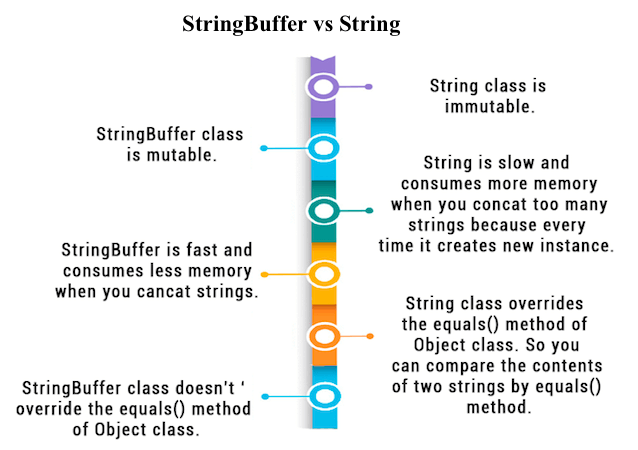
The ensureCapacity() method of StringBuilder class ensures that the given capacity is the minimum to the current capacity. If it is greater than the current capacity, it increases the capacity by (oldcapacity\*2)+2. For example if your current capacity is 16, it will be (16\*2)+2=34.

1. **class** StringBuilderExample7{
2. **public** **static** **void** main(String args[]){
3. StringBuilder sb=**new** StringBuilder();
4. System.out.println(sb.capacity());//default 16
5. sb.append("Hello");
6. System.out.println(sb.capacity());//now 16
7. sb.append("java is my favourite language");
8. System.out.println(sb.capacity());//now (16\*2)+2=34 i.e (oldcapacity\*2)+2
9. sb.ensureCapacity(10);//now no change
10. System.out.println(sb.capacity());//now 34
11. sb.ensureCapacity(50);//now (34\*2)+2
12. System.out.println(sb.capacity());//now 70
13. }
14. }

# Difference between String and StringBuffer

There are many differences between String and StringBuffer. A list of differences between String and StringBuffer are given below:

|  |  |  |
| --- | --- | --- |
| **No.** | **String** | **StringBuffer** |
| 1) | String class is immutable. | StringBuffer class is mutable. |
| 2) | String is slow and consumes more memory when you concat too many strings because every time it creates new instance. | StringBuffer is fast and consumes less memory when you cancat strings. |
| 3) | String class overrides the equals() method of Object class. So you can compare the contents of two strings by equals() method. | StringBuffer class doesn't override the equals() method of Object class. |



## Performance Test of String and StringBuffer

1. **public** **class** ConcatTest{
2. **public** **static** String concatWithString()    {
3. String t = "Java";
4. **for** (**int** i=0; i<10000; i++){
5. t = t + "Tpoint";
6. }
7. **return** t;
8. }
9. **public** **static** String concatWithStringBuffer(){
10. StringBuffer sb = **new** StringBuffer("Java");
11. **for** (**int** i=0; i<10000; i++){
12. sb.append("Tpoint");
13. }
14. **return** sb.toString();
15. }
16. **public** **static** **void** main(String[] args){
17. **long** startTime = System.currentTimeMillis();
18. concatWithString();
19. System.out.println("Time taken by Concating with String: "+(System.currentTimeMillis()-startTime)+"ms");
20. startTime = System.currentTimeMillis();
21. concatWithStringBuffer();
22. System.out.println("Time taken by Concating with  StringBuffer: "+(System.currentTimeMillis()-startTime)+"ms");
23. }
24. }

Time taken by Concating with String: 578ms

Time taken by Concating with StringBuffer: 0ms

## String and StringBuffer HashCode Test

As you can see in the program given below, String returns new hashcode value when you concat string but StringBuffer returns same.

1. **public** **class** InstanceTest{
2. **public** **static** **void** main(String args[]){
3. System.out.println("Hashcode test of String:");
4. String str="java";
5. System.out.println(str.hashCode());
6. str=str+"tpoint";
7. System.out.println(str.hashCode());
9. System.out.println("Hashcode test of StringBuffer:");
10. StringBuffer sb=**new** StringBuffer("java");
11. System.out.println(sb.hashCode());
12. sb.append("tpoint");
13. System.out.println(sb.hashCode());
14. }
15. }

Hashcode test of String:

3254818

229541438

Hashcode test of StringBuffer:

118352462

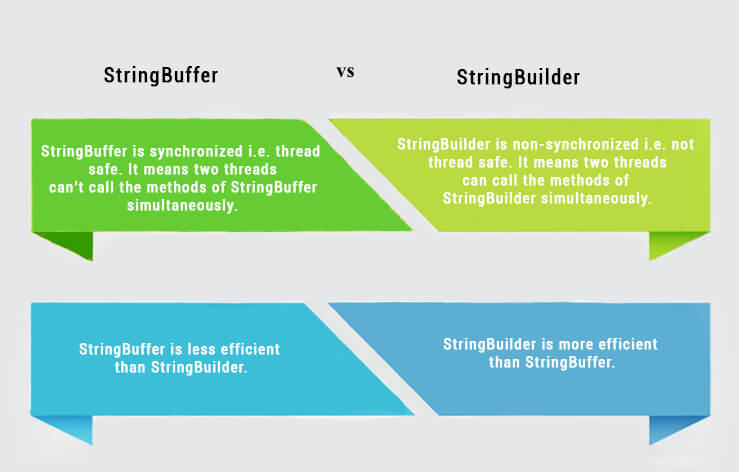
118352462

# Difference between StringBuffer and StringBuilder

Java provides three classes to represent a sequence of characters: String, StringBuffer, and StringBuilder. The String class is an immutable class whereas StringBuffer and StringBuilder classes are mutable. There are many differences between StringBuffer and StringBuilder. The StringBuilder class is introduced since JDK 1.5.

A list of differences between StringBuffer and StringBuilder are given below:

|  |  |  |
| --- | --- | --- |
| **No.** | **StringBuffer** | **StringBuilder** |
| 1) | StringBuffer is synchronized i.e. thread safe. It means two threads can't call the methods of StringBuffer simultaneously. | StringBuilder is non-synchronized i.e. not thread safe. It means two threads can call the methods of StringBuilder simultaneously. |
| 2) | StringBuffer is less efficient than StringBuilder. | StringBuilder is more efficient than StringBuffer. |



## StringBuffer Example

1. //Java Program to demonstrate the use of StringBuffer class.
2. **public** **class** BufferTest{
3. **public** **static** **void** main(String[] args){
4. StringBuffer buffer=**new** StringBuffer("hello");
5. buffer.append("java");
6. System.out.println(buffer);
7. }
8. }

hellojava

## StringBuilder Example

1. //Java Program to demonstrate the use of StringBuilder class.
2. **public** **class** BuilderTest{
3. **public** **static** **void** main(String[] args){
4. StringBuilder builder=**new** StringBuilder("hello");
5. builder.append("java");
6. System.out.println(builder);
7. }
8. }

hellojava

## Performance Test of StringBuffer and StringBuilder

Let's see the code to check the performance of StringBuffer and StringBuilder classes.

1. //Java Program to demonstrate the performance of StringBuffer and StringBuilder classes.
2. **public** **class** ConcatTest{
3. **public** **static** **void** main(String[] args){
4. **long** startTime = System.currentTimeMillis();
5. StringBuffer sb = **new** StringBuffer("Java");
6. **for** (**int** i=0; i<10000; i++){
7. sb.append("Tpoint");
8. }
9. System.out.println("Time taken by StringBuffer: " + (System.currentTimeMillis() - startTime) + "ms");
10. startTime = System.currentTimeMillis();
11. StringBuilder sb2 = **new** StringBuilder("Java");
12. **for** (**int** i=0; i<10000; i++){
13. sb2.append("Tpoint");
14. }
15. System.out.println("Time taken by StringBuilder: " + (System.currentTimeMillis() - startTime) + "ms");
16. }
17. }

Time taken by StringBuffer: 16ms

Time taken by StringBuilder: 0ms

# How to create Immutable class?

There are many immutable classes like String, Boolean, Byte, Short, Integer, Long, Float, Double etc. In short, all the wrapper classes and String class is immutable. We can also create immutable class by creating final class that have final data members as the example given below:

### Example to create Immutable class

|  |
| --- |
| In this example, we have created a final class named Employee. It have one final datamember, a parameterized constructor and getter method. |

1. **public** **final** **class** Employee{
2. **final** String pancardNumber;
4. **public** Employee(String pancardNumber){
5. **this**.pancardNumber=pancardNumber;
6. }
8. **public** String getPancardNumber(){
9. **return** pancardNumber;
10. }
12. }

The above class is immutable because:

* The instance variable of the class is final i.e. we cannot change the value of it after creating an object.
* The class is final so we cannot create the subclass.
* There is no setter methods i.e. we have no option to change the value of the instance variable.

These points makes this class as immutable.

# Java toString() method

If you want to represent any object as a string, **toString() method** comes into existence.

The toString() method returns the string representation of the object.

If you print any object, java compiler internally invokes the toString() method on the object. So overriding the toString() method, returns the desired output, it can be the state of an object etc. depends on your implementation.

## Advantage of Java toString() method

By overriding the toString() method of the Object class, we can return values of the object, so we don't need to write much code.

### Understanding problem without toString() method

Let's see the simple code that prints reference.

1. **class** Student{
2. **int** rollno;
3. String name;
4. String city;
6. Student(**int** rollno, String name, String city){
7. **this**.rollno=rollno;
8. **this**.name=name;
9. **this**.city=city;
10. }
12. **public** **static** **void** main(String args[]){
13. Student s1=**new** Student(101,"Raj","lucknow");
14. Student s2=**new** Student(102,"Vijay","ghaziabad");
16. System.out.println(s1);//compiler writes here s1.toString()
17. System.out.println(s2);//compiler writes here s2.toString()
18. }
19. }

Output:Student@1fee6fc

Student@1eed786

|  |  |
| --- | --- |
| As you can see in the above example, printing s1 and s2 prints the hashcode values of the objects but I want to print the values of these objects. Since java compiler internally calls toString() method, overriding this method will return the specified values. Let's understand it with the example given below: |  |

## Example of Java toString() method

Now let's see the real example of toString() method.

1. **class** Student{
2. **int** rollno;
3. String name;
4. String city;
6. Student(**int** rollno, String name, String city){
7. **this**.rollno=rollno;
8. **this**.name=name;
9. **this**.city=city;
10. }
12. **public** String toString(){//overriding the toString() method
13. **return** rollno+" "+name+" "+city;
14. }
15. **public** **static** **void** main(String args[]){
16. Student s1=**new** Student(101,"Raj","lucknow");
17. Student s2=**new** Student(102,"Vijay","ghaziabad");
19. System.out.println(s1);//compiler writes here s1.toString()
20. System.out.println(s2);//compiler writes here s2.toString()
21. }
22. }

[download this example of toString method](https://static.javatpoint.com/src/string/tostring.zip)

Output:101 Raj lucknow

102 Vijay ghaziabad

# StringTokenizer in Java

# The java.util.StringTokenizer class allows you to break a string into tokens. It is simple way to break string.

It doesn't provide the facility to differentiate numbers, quoted strings, identifiers etc. like StreamTokenizer class. We will discuss about the StreamTokenizer class in I/O chapter.

#### Constructors of StringTokenizer class

There are 3 constructors defined in the StringTokenizer class.

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| StringTokenizer(String str) | creates StringTokenizer with specified string. |
| StringTokenizer(String str, String delim) | creates StringTokenizer with specified string and delimeter. |
| StringTokenizer(String str, String delim, boolean returnValue) | creates StringTokenizer with specified string, delimeter and returnValue. If return value is true, delimiter characters are considered to be tokens. If it is false, delimiter characters serve to separate tokens. |

#### Methods of StringTokenizer class

The 6 useful methods of StringTokenizer class are as follows:

|  |  |
| --- | --- |
| **Public method** | **Description** |
| boolean hasMoreTokens() | checks if there is more tokens available. |
| String nextToken() | returns the next token from the StringTokenizer object. |
| String nextToken(String delim) | returns the next token based on the delimeter. |
| boolean hasMoreElements() | same as hasMoreTokens() method. |
| Object nextElement() | same as nextToken() but its return type is Object. |
| int countTokens() | returns the total number of tokens. |

### Simple example of StringTokenizer class

Let's see the simple example of StringTokenizer class that tokenizes a string "my name is khan" on the basis of whitespace.

1. **import** java.util.StringTokenizer;
2. **public** **class** Simple{
3. **public** **static** **void** main(String args[]){
4. StringTokenizer st = **new** StringTokenizer("my name is khan"," ");
5. **while** (st.hasMoreTokens()) {
6. System.out.println(st.nextToken());
7. }
8. }
9. }

Output:my

name

is

khan

### Example of nextToken(String delim) method of StringTokenizer class

1. **import** java.util.\*;
3. **public** **class** Test {
4. **public** **static** **void** main(String[] args) {
5. StringTokenizer st = **new** StringTokenizer("my,name,is,khan");
7. // printing next token
8. System.out.println("Next token is : " + st.nextToken(","));
9. }
10. }

Output:Next token is : my

#### StringTokenizer class is deprecated now. It is recommended to use split() method of String class or regex (Regular Expression).

# Java String FAQs or Interview Questions

A list of top Java String FAQs (Frequently Asked Questions) or interview questions are given below. These questions can be asked by the interviewer.

### 1) How many objects will be created in the following code?

String s1="javatpoint";

String s2="javatpoint";

**Answer:** Only one.

### 2) What is the difference between equals() method and == operator?

The equals() method matches content of the strings whereas == operator matches object or reference of the strings.

### 3) Is String class final?

**Answer:** Yes.

# How to reverse String in Java

There are many ways to reverse String in Java. We can reverse String using StringBuffer, StringBuilder, iteration etc. Let's see the ways to reverse String in Java.

## 1) By StringBuilder / StringBuffer

*File: StringFormatter.java*

1. **public** **class** StringFormatter {
2. **public** **static** String reverseString(String str){
3. StringBuilder sb=**new** StringBuilder(str);
4. sb.reverse();
5. **return** sb.toString();
6. }
7. }

*File: TestStringFormatter.java*

1. **public** **class** TestStringFormatter {
2. **public** **static** **void** main(String[] args) {
3. System.out.println(StringFormatter.reverseString("my name is khan"));
4. System.out.println(StringFormatter.reverseString("I am sonoo jaiswal"));
5. }
6. }

Output:

nahk si eman ym

lawsiaj oonos ma I

## 2) By Reverse Iteration

*File: StringFormatter.java*

1. **public** **class** StringFormatter {
2. **public** **static** String reverseString(String str){
3. **char** ch[]=str.toCharArray();
4. String rev="";
5. **for**(**int** i=ch.length-1;i>=0;i--){
6. rev+=ch[i];
7. }
8. **return** rev;
9. }
10. }

*File: TestStringFormatter.java*

1. **public** **class** TestStringFormatter {
2. **public** **static** **void** main(String[] args) {
3. System.out.println(StringFormatter.reverseString("my name is khan"));
4. System.out.println(StringFormatter.reverseString("I am sonoo jaiswal"));
5. }
6. }

Output:

nahk si eman ym

lawsiaj oonos ma I

***Java OOPs Misc***

Object class in Java

The **Object class** is the parent class of all the classes in java by default. In other words, it is the topmost class of java.

The Object class is beneficial if you want to refer any object whose type you don't know. Notice that parent class reference variable can refer the child class object, know as upcasting.

Let's take an example, there is getObject() method that returns an object but it can be of any type like Employee,Student etc, we can use Object class reference to refer that object. For example:

1. Object obj=getObject();//we don't know what object will be returned from this method

The Object class provides some common behaviors to all the objects such as object can be compared, object can be cloned, object can be notified etc.



Methods of Object class

|  |
| --- |
| The Object class provides many methods. They are as follows: |

|  |  |
| --- | --- |
| **Method** | **Description** |
| public final Class getClass() | returns the Class class object of this object. The Class class can further be used to get the metadata of this class. |
| public int hashCode() | returns the hashcode number for this object. |
| public boolean equals(Object obj) | compares the given object to this object. |
| protected Object clone() throws CloneNotSupportedException | creates and returns the exact copy (clone) of this object. |
| public String toString() | returns the string representation of this object. |
| public final void notify() | wakes up single thread, waiting on this object's monitor. |
| public final void notifyAll() | wakes up all the threads, waiting on this object's monitor. |
| public final void wait(long timeout)throws InterruptedException | causes the current thread to wait for the specified milliseconds, until another thread notifies (invokes notify() or notifyAll() method). |
| public final void wait(long timeout,int nanos)throws InterruptedException | causes the current thread to wait for the specified milliseconds and nanoseconds, until another thread notifies (invokes notify() or notifyAll() method). |
| public final void wait()throws InterruptedException | causes the current thread to wait, until another thread notifies (invokes notify() or notifyAll() method). |
| protected void finalize()throws Throwable | is invoked by the garbage collector before object is being garbage collected. |

We will have the detailed learning of these methods in next chapters.

Java Math class

Java Math class provides several methods to work on math calculations like min(), max(), avg(), sin(), cos(), tan(), round(), ceil(), floor(), abs() etc.

Unlike some of the StrictMath class numeric methods, all implementations of the equivalent function of Math class can't define to return the bit-for-bit same results. This relaxation permits implementation with better-performance where strict reproducibility is not required.

If the size is int or long and the results overflow the range of value, the methods addExact(), subtractExact(), multiplyExact(), and toIntExact() throw an ArithmeticException.

For other arithmetic operations like increment, decrement, divide, absolute value, and negation overflow occur only with a specific minimum or maximum value. It should be checked against the maximum and minimum value as appropriate.

Example 1

1. **public** **class** JavaMathExample1
2. {
3. **public** **static** **void** main(String[] args)
4. {
5. **double** x = 28;
6. **double** y = 4;
8. // return the maximum of two numbers
9. System.out.println("Maximum number of x and y is: " +Math.max(x, y));
11. // return the square root of y
12. System.out.println("Square root of y is: " + Math.sqrt(y));
14. //returns 28 power of 4 i.e. 28\*28\*28\*28
15. System.out.println("Power of x and y is: " + Math.pow(x, y));
17. // return the logarithm of given value
18. System.out.println("Logarithm of x is: " + Math.log(x));
19. System.out.println("Logarithm of y is: " + Math.log(y));
21. // return the logarithm of given value when base is 10
22. System.out.println("log10 of x is: " + Math.log10(x));
23. System.out.println("log10 of y is: " + Math.log10(y));
25. // return the log of x + 1
26. System.out.println("log1p of x is: " +Math.log1p(x));
28. // return a power of 2
29. System.out.println("exp of a is: " +Math.exp(x));
31. // return (a power of 2)-1
32. System.out.println("expm1 of a is: " +Math.expm1(x));
33. }
34. }

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=JavaMathExample1)

**Output:**

Maximum number of x and y is: 28.0

Square root of y is: 2.0

Power of x and y is: 614656.0

Logarithm of x is: 3.332204510175204

Logarithm of y is: 1.3862943611198906

log10 of x is: 1.4471580313422192

log10 of y is: 0.6020599913279624

log1p of x is: 3.367295829986474

exp of a is: 1.446257064291475E12

expm1 of a is: 1.446257064290475E12

Example 2

1. **public** **class** JavaMathExample2
2. {
3. **public** **static** **void** main(String[] args)
4. {
5. **double** a = 30;
7. // converting values to radian
8. **double** b = Math.toRadians(a);
10. // return the trigonometric sine of a
11. System.out.println("Sine value of a is: " +Math.sin(a));
13. // return the trigonometric cosine value of a
14. System.out.println("Cosine value of a is: " +Math.cos(a));
16. // return the trigonometric tangent value of a
17. System.out.println("Tangent value of a is: " +Math.tan(a));
19. // return the trigonometric arc sine of a
20. System.out.println("Sine value of a is: " +Math.asin(a));
22. // return the trigonometric arc cosine value of a
23. System.out.println("Cosine value of a is: " +Math.acos(a));
25. // return the trigonometric arc tangent value of a
26. System.out.println("Tangent value of a is: " +Math.atan(a));
28. // return the hyperbolic sine of a
29. System.out.println("Sine value of a is: " +Math.sinh(a));
31. // return the hyperbolic cosine value of a
32. System.out.println("Cosine value of a is: " +Math.cosh(a));
34. // return the hyperbolic tangent value of a
35. System.out.println("Tangent value of a is: " +Math.tanh(a));
36. }
37. }

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=JavaMathExample2)

**Output:**

Sine value of a is: -0.9880316240928618

Cosine value of a is: 0.15425144988758405

Tangent value of a is: -6.405331196646276

Sine value of a is: NaN

Cosine value of a is: NaN

Tangent value of a is: 1.5374753309166493

Sine value of a is: 5.343237290762231E12

Cosine value of a is: 5.343237290762231E12

Tangent value of a is: 1.0

Java Math Methods

The **java.lang.Math** class contains various methods for performing basic numeric operations such as the logarithm, cube root, and trigonometric functions etc. The various java math methods are as follows:

Basic Math methods

|  |  |
| --- | --- |
| **Method** | **Description** |
| [Math.abs()](https://www.javatpoint.com/java-math-abs-method) | It will return the Absolute value of the given value. |
| [Math.max()](https://www.javatpoint.com/java-math-max-method) | It returns the Largest of two values. |
| [Math.min()](https://www.javatpoint.com/java-math-min-method) | It is used to return the Smallest of two values. |
| [Math.round()](https://www.javatpoint.com/java-math-round-method) | It is used to round of the decimal numbers to the nearest value. |
| [Math.sqrt()](https://www.javatpoint.com/java-math-sqrt-method) | It is used to return the square root of a number. |
| [Math.cbrt()](https://www.javatpoint.com/java-math-cbrt-method) | It is used to return the cube root of a number. |
| [Math.pow()](https://www.javatpoint.com/java-math-pow-method) | It returns the value of first argument raised to the power to second argument. |
| [Math.signum()](https://www.javatpoint.com/java-math-signum-method) | It is used to find the sign of a given value. |
| [Math.ceil()](https://www.javatpoint.com/java-math-ceil-method) | It is used to find the smallest integer value that is greater than or equal to the argument or mathematical integer. |
| [Math.copySign()](https://www.javatpoint.com/java-math-copysign-method) | It is used to find the Absolute value of first argument along with sign specified in second argument. |
| [Math.nextAfter()](https://www.javatpoint.com/java-math-nextafter-method) | It is used to return the floating-point number adjacent to the first argument in the direction of the second argument. |
| [Math.nextUp()](https://www.javatpoint.com/java-math-nextup-method) | It returns the floating-point value adjacent to d in the direction of positive infinity. |
| [Math.nextDown()](https://www.javatpoint.com/java-math-nextdown-method) | It returns the floating-point value adjacent to d in the direction of negative infinity. |
| [Math.floor()](https://www.javatpoint.com/java-math-floor-method) | It is used to find the largest integer value which is less than or equal to the argument and is equal to the mathematical integer of a double value. |
| [Math.floorDiv()](https://www.javatpoint.com/java-math-floordiv-method) | It is used to find the largest integer value that is less than or equal to the algebraic quotient. |
| [Math.random()](https://www.javatpoint.com/java-math-random-method) | It returns a double value with a positive sign, greater than or equal to 0.0 and less than 1.0. |
| [Math.rint()](https://www.javatpoint.com/java-math-rint-method) | It returns the double value that is closest to the given argument and equal to mathematical integer. |
| [Math.hypot()](https://www.javatpoint.com/java-math-hypot-method) | It returns sqrt(x2 +y2) without intermediate overflow or underflow. |
| [Math.ulp()](https://www.javatpoint.com/java-math-ulp-method) | It returns the size of an ulp of the argument. |
| [Math.getExponent()](https://www.javatpoint.com/java-math-getexponent-method) | It is used to return the unbiased exponent used in the representation of a value. |
| [Math.IEEEremainder()](https://www.javatpoint.com/java-math-ieeeremainder-method) | It is used to calculate the remainder operation on two arguments as prescribed by the IEEE 754 standard and returns value. |
| [Math.addExact()](https://www.javatpoint.com/java-math-addexact-method) | It is used to return the sum of its arguments, throwing an exception if the result overflows an int or long. |
| [Math.subtractExact()](https://www.javatpoint.com/java-math-subtractexact-method) | It returns the difference of the arguments, throwing an exception if the result overflows an int. |
| [Math.multiplyExact()](https://www.javatpoint.com/java-math-multiplyexact-method) | It is used to return the product of the arguments, throwing an exception if the result overflows an int or long. |
| [Math.incrementExact()](https://www.javatpoint.com/java-math-incrementexact-method) | It returns the argument incremented by one, throwing an exception if the result overflows an int. |
| [Math.decrementExact()](https://www.javatpoint.com/java-math-decrementexact-method) | It is used to return the argument decremented by one, throwing an exception if the result overflows an int or long. |
| [Math.negateExact()](https://www.javatpoint.com/java-math-negateexact-method) | It is used to return the negation of the argument, throwing an exception if the result overflows an int or long. |
| [Math.toIntExact()](https://www.javatpoint.com/java-math-tointexact-method) | It returns the value of the long argument, throwing an exception if the value overflows an int. |

Logarithmic Math Methods

|  |  |
| --- | --- |
| **Method** | **Description** |
| [Math.log()](https://www.javatpoint.com/java-math-log-method) | It returns the natural logarithm of a double value. |
| [Math.log10()](https://www.javatpoint.com/java-math-log10-method) | It is used to return the base 10 logarithm of a double value. |
| [Math.log1p()](https://www.javatpoint.com/java-math-log1p-method) | It returns the natural logarithm of the sum of the argument and 1. |
| [Math.exp()](https://www.javatpoint.com/java-math-exp-method) | It returns E raised to the power of a double value, where E is Euler's number and it is approximately equal to 2.71828. |
| [Math.expm1()](https://www.javatpoint.com/java-math-expm1-method) | It is used to calculate the power of E and subtract one from it. |

Trigonometric Math Methods

|  |  |
| --- | --- |
| **Method** | **Description** |
| [Math.sin()](https://www.javatpoint.com/java-math-sin-method) | It is used to return the trigonometric Sine value of a Given double value. |
| [Math.cos()](https://www.javatpoint.com/java-math-cos-method) | It is used to return the trigonometric Cosine value of a Given double value. |
| [Math.tan()](https://www.javatpoint.com/java-math-tan-method) | It is used to return the trigonometric Tangent value of a Given double value. |
| [Math.asin()](https://www.javatpoint.com/java-math-asin-method) | It is used to return the trigonometric Arc Sine value of a Given double value |
| [Math.acos()](https://www.javatpoint.com/java-math-acos-method) | It is used to return the trigonometric Arc Cosine value of a Given double value. |
| [Math.atan()](https://www.javatpoint.com/java-math-atan-method) | It is used to return the trigonometric Arc Tangent value of a Given double value. |

Hyperbolic Math Methods

|  |  |
| --- | --- |
| **Method** | **Description** |
| [Math.sinh()](https://www.javatpoint.com/java-math-sinh-method) | It is used to return the trigonometric Hyperbolic Cosine value of a Given double value. |
| [Math.cosh()](https://www.javatpoint.com/java-math-cosh-method) | It is used to return the trigonometric Hyperbolic Sine value of a Given double value. |
| [Math.tanh()](https://www.javatpoint.com/java-math-tanh-method) | It is used to return the trigonometric Hyperbolic Tangent value of a Given double value. |

Angular Math Methods

|  |  |
| --- | --- |
| **Method** | **Description** |
| [Math.toDegrees](https://www.javatpoint.com/java-math-todegrees-method) | It is used to convert the specified Radians angle to equivalent angle measured in Degrees. |
| [Math.toRadians](https://www.javatpoint.com/java-math-toradians-method) | It is used to convert the specified Degrees angle to equivalent angle measured in Radians. |

***Wrapper classes in Java***

|  |
| --- |
| ==================================Shiv nath sir===============================  Wrapper Classes  ==================  int a = 5;  float b = 23.5;  class someClass  {  //Performs some operation  // But it does not work with primitive data  // It works only with objects  void add(object x, object y)  {  //sop(x+y);  }  }  eight primitive types  byte  short  int  long  float  double  char  boolean  For each primitive type there is a wrapper class to wrap that primitive type into an object.  byte => Byte  short => Short  int => Integer  long => Long  likewise Float, Double, Boolean and Character wrapper classes are there.  Number is an abstract class and it is the superclass for the classes Byte, Short, Integer, Long, Float and Double.  Number contains abstract methods:  byte byteValue();  int intValue();  double d = 23.5;  Double dOb = new Double (23.5); // Will wrap double data into object and will create a Double object  Double dOb = new Double (d);  sop(dOb.intValue()); // Prints 23  class Double extends Number  {  double data;  int intValue()  {  // will return int equivalent value of double data  }  byte byteValue()  {  }  double doubleValue()  {  }  }  To create wrapper class object  ===============================  double d = 23.5;  Double dOb = new Double(23.5);  Double dOb = new Double("23.5");  Double dob = new Double(d)  int i = 23;  Integer iOb = new Integer(23);  Integer iOb = new Integer("23");  Integer iOb = new Integer(i)  Double dOb = new Double("Hello"); //NumberFormatException  Double dOb1 = new Double("23.5");  Double dOb2 = new Double("23.5");  int res = dOb1.compareTo(dOb2); // if returns zero, means both are same  System.out.println(res) ; // prints 0  dOb1.compareTo(dOb2);  // if dOb1 == dOb2 then it will return 0  // if dOb1 < dOb2 then it will return -1  // if dOb1 > dOb2 then it will return 1  We can not compare two different wrapper type like Integer and Double objects.  String str = "23.5";  double d = Double.parseDouble(str);  sop(d) ; // will print 23.5  str+5;  double d = 23.5;  ========================================================  iOb1.compareTo(new Integer(dOb1.intValue()));  Double dOb1 = new Double("23.5");  String dStr1 = dOb1.toString(); // will give String "23.5" not a number  dStr1 \* 5; //Error, we can not perform artihmetic operation with String type.  =============================================================  Creating a Double object from the given String value  String strVal = "23.5";  Double dOb = new Double(strVal);  Double dOb = Double.valueOf(strVal);  =================================================================  Boolean blOb = new Boolean(true);  Boolean blOb = new Boolean("true");  sop(blOb.toString()) ; // print true  ===============================================================  char ch = 'X';  Character chOb = new Character('X');  Character chOb = new Character(ch);  chOb.toString(); // will give "X" - It is not a charcter, it is a String  Integer iOb = new Integer(10); // Explicitly boxing  int x = iOb.intValue(); // Explicitly unboxing  Integer iOb = 10; // Automatic boxing  int x = iOb; // Automatic unboxing  int a = 10;  myFun(iOb);  void myFun(int i)  {  }  Integer = int // Automatic boxing  int = Integer // Automatic unboxing  int a = 5;  double b = 5.0;  int res = a \* b;  Integer a = 5;  Double b = 5.0;  int res = a \* b; |

* The **wrapper class in Java** provides the mechanism *to convert primitive into object and object into primitive*.
* Since J2SE 5.0, **autoboxing** and **unboxing** feature convert primitives into objects and objects into primitives automatically. The automatic conversion of primitive into an object is known as autoboxing and vice-versa unboxing.

Use of Wrapper classes in Java

Java is an object-oriented programming language, so we need to deal with objects many times like in Collections, Serialization, Synchronization, etc. Let us see the different scenarios, where we need to use the wrapper classes.

* **Change the value in Method:** Java supports only call by value. So, if we pass a primitive value, it will not change the original value. But, if we convert the primitive value in an object, it will change the original value.
* **Serialization:** We need to convert the objects into streams to perform the serialization. If we have a primitive value, we can convert it in objects through the wrapper classes.
* **Synchronization:** Java synchronization works with objects in Multithreading.
* **java.util package:** The java.util package provides the utility classes to deal with objects.
* **Collection Framework:** Java collection framework works with objects only. All classes of the collection framework (ArrayList, LinkedList, Vector, HashSet, LinkedHashSet, TreeSet, PriorityQueue, ArrayDeque, etc.) deal with objects only.

The eight classes of the *java.lang* package are known as wrapper classes in Java. The list of eight wrapper classes are given below:

|  |  |
| --- | --- |
| **Primitive Type** | **Wrapper class** |
| boolean | [Boolean](https://www.javatpoint.com/java-boolean) |
| char | [Character](https://www.javatpoint.com/post/java-character) |
| byte | [Byte](https://www.javatpoint.com/java-byte) |
| short | [Short](https://www.javatpoint.com/java-short) |
| int | [Integer](https://www.javatpoint.com/java-integer) |
| long | [Long](https://www.javatpoint.com/java-long) |
| float | [Float](https://www.javatpoint.com/java-float) |
| double | [Double](https://www.javatpoint.com/java-double) |

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.

Since Java 5, we do not need to use the valueOf() method of wrapper classes to convert the primitive into objects.

**Wrapper class Example: Primitive to Wrapper**

1. //Java program to convert primitive into objects
2. //Autoboxing example of int to Integer
3. **public** **class** WrapperExample1{
4. **public** **static** **void** main(String args[]){
5. //Converting int into Integer
6. **int** a=20;
7. Integer i=Integer.valueOf(a);//converting int into Integer explicitly
8. Integer j=a;//autoboxing, now compiler will write Integer.valueOf(a) internally
10. System.out.println(a+" "+i+" "+j);
11. }}

Output:

20 20 20

Unboxing

The automatic conversion of wrapper type into its corresponding primitive type is known as unboxing. It is the reverse process of autoboxing. Since Java 5, we do not need to use the intValue() method of wrapper classes to convert the wrapper type into primitives.

**Wrapper class Example: Wrapper to Primitive**

1. //Java program to convert object into primitives
2. //Unboxing example of Integer to int
3. **public** **class** WrapperExample2{
4. **public** **static** **void** main(String args[]){
5. //Converting Integer to int
6. Integer a=**new** Integer(3);
7. **int** i=a.intValue();//converting Integer to int explicitly
8. **int** j=a;//unboxing, now compiler will write a.intValue() internally
10. System.out.println(a+" "+i+" "+j);
11. }}

Output:

3 3 3

Java Wrapper classes Example

1. //Java Program to convert all primitives into its corresponding
2. //wrapper objects and vice-versa
3. **public** **class** WrapperExample3{
4. **public** **static** **void** main(String args[]){
5. **byte** b=10;
6. **short** s=20;
7. **int** i=30;
8. **long** l=40;
9. **float** f=50.0F;
10. **double** d=60.0D;
11. **char** c='a';
12. **boolean** b2=**true**;
14. //Autoboxing: Converting primitives into objects
15. Byte byteobj=b;
16. Short shortobj=s;
17. Integer intobj=i;
18. Long longobj=l;
19. Float floatobj=f;
20. Double doubleobj=d;
21. Character charobj=c;
22. Boolean boolobj=b2;
24. //Printing objects
25. System.out.println("---Printing object values---");
26. System.out.println("Byte object: "+byteobj);
27. System.out.println("Short object: "+shortobj);
28. System.out.println("Integer object: "+intobj);
29. System.out.println("Long object: "+longobj);
30. System.out.println("Float object: "+floatobj);
31. System.out.println("Double object: "+doubleobj);
32. System.out.println("Character object: "+charobj);
33. System.out.println("Boolean object: "+boolobj);
35. //Unboxing: Converting Objects to Primitives
36. **byte** bytevalue=byteobj;
37. **short** shortvalue=shortobj;
38. **int** intvalue=intobj;
39. **long** longvalue=longobj;
40. **float** floatvalue=floatobj;
41. **double** doublevalue=doubleobj;
42. **char** charvalue=charobj;
43. **boolean** boolvalue=boolobj;
45. //Printing primitives
46. System.out.println("---Printing primitive values---");
47. System.out.println("byte value: "+bytevalue);
48. System.out.println("short value: "+shortvalue);
49. System.out.println("int value: "+intvalue);
50. System.out.println("long value: "+longvalue);
51. System.out.println("float value: "+floatvalue);
52. System.out.println("double value: "+doublevalue);
53. System.out.println("char value: "+charvalue);
54. System.out.println("boolean value: "+boolvalue);
55. }}

Output:

---Printing object values---

Byte object: 10

Short object: 20

Integer object: 30

Long object: 40

Float object: 50.0

Double object: 60.0

Character object: a

Boolean object: true

---Printing primitive values---

byte value: 10

short value: 20

int value: 30

long value: 40

float value: 50.0

double value: 60.0

char value: a

boolean value: true

Custom Wrapper class in Java

Java Wrapper classes wrap the primitive data types, that is why it is known as wrapper classes. We can also create a class which wraps a primitive data type. So, we can create a custom wrapper class in Java.

1. //Creating the custom wrapper class
2. **class** Javatpoint{
3. **private** **int** i;
4. Javatpoint(){}
5. Javatpoint(**int** i){
6. **this**.i=i;
7. }
8. **public** **int** getValue(){
9. **return** i;
10. }
11. **public** **void** setValue(**int** i){
12. **this**.i=i;
13. }
14. @Override
15. **public** String toString() {
16. **return** Integer.toString(i);
17. }
18. }
19. //Testing the custom wrapper class
20. **public** **class** TestJavatpoint{
21. **public** **static** **void** main(String[] args){
22. Javatpoint j=**new** Javatpoint(10);
23. System.out.println(j);
24. }}

Output:

10

***Call by Value and Call by Reference in Java::***

***Call by Value::-***

|  |  |
| --- | --- |
| * Call by Value means calling a method with a parameter as value. * The changes being done in the called method, is not affected in the calling method. * In call by value, the modification done to the parameter passed does not reflect in the caller's scope while in the call by reference, the modification done to the parameter passed are persistent and changes are reflected in the caller's scope. * In call by value, the modification done to the parameter passed does not reflect in the caller's scope while in the call by reference, the modification done to the parameter passed are persistent and changes are reflected in the caller's scope. * While Call by Reference means calling a method with a parameter as a reference. Through this, the argument reference is passed to the parameter.   ***Call by Reference:***   * In case of call by reference original value is changed if we made changes in the called method. If we pass object in place of any primitive value, original value will be changed  |  | | --- | | ===============================Shiv nath sir=====================================  //Call By Value - Primitive Data is passed by Value//Changes will not reflect in actual Data//Call by Reference - Object is passed by Reference but that reference is passed by value//If we want to make change using this reference ,that will also be reflected in the actual object whose reference has been passed  //or reflect in actual data  Faculty====  /\*-=====================================  function calling::::call by value & call by value reference  -=======================================  \*/  package CBVCBR;  class First  {  int a; //instance variable  int b;    First(int a,int b)  {  this.a =a; //local variable  this.b =b;  }  }  class Second  {  int b;  void change(int x, int y)  {  x = x + 1; //x=6  y =y + 1; //y=11  }  void change(First first)  {  first.a = first.a+1; //51  first.b = first.b+1; //101  }  }  class FunctionCallDemo  {  public static void main(String args[])  {  int p =5;  int q= 10;  System.out.println(p); //print 5  System.out.println(q); //print 10  System.out.println();  Second s = new Second();  s.change(p,q);  System.out.println(p); //print 5  System.out.println(q); //print 10    System.out.println();    First f = new First(50,100);  System.out.println(f.a); //print 50  System.out.println(f.b); //print100    s.change(f);  System.out.println();    System.out.println(f.a); //print 51  System.out.println(f.b); //print101  }  } | |

***Casting Referenced Data Types***

|  |
| --- |
| =================================Shiv nath sir================================  Casting Referenced Data Types  ===============================================  Converting a class type into another class type is possible through casting.  Casting is possible between parent and child class object references.  casting from child to parent is called upcasting (widening conversion, generalisation)  casting from parent to child is called downcasting (narrowing conversion, specialisation)  double d = 23.5;  int i = (int) d;  int j = 20;  d = j;  double = int (widening conversion, there we don't need to do any explicit casting)  int = double (narrowing conversion, there we need to do explicit casting)  class First  {  void fun()  {  System.out.println("fun of First");  }  void fun1()  {  System.out.println("fun1 of First");  }  }  class Second extends First  {  void fun()  {  System.out.println("fun of Second");  }  void fun2()  {  System.out.println("fun2 of Second");  }    }  class Third  {  void fun()  {  System.out.println("fun of Third");  }  }  class CastingDemo  {  public static void main(String args[])  {  First f;  Second s;  //f = new First();  //s = (Second)f; //child = parent; Here explicit casting is must  // Error at runtime because f contains reference of object of First    f = new Second(); //Parent = child  s = (Second)f; //Works, beacuase at runtime f contains reference of object of Second  s.fun();  s.fun1();  s.fun2();    //f.fun2(); //Compiler will give error as First does not contain function fun2()  ((Second)f).fun2();  }  }  compilation -> compiler checks type of reference  execution -> jvm checks actual object's reference  At runtime jvm ensures that child class reference variable should not contain parent class' object reference.  child class reference variable can not contain parent class' object's reference  Any class can contain referene of its own object or object of it's child class. |

# Multithreading in Java

|  |
| --- |
| ==========================================Shiv nath sir===========================  /\*  =================Multitasking========  To perform multiple task simultaneusy  ==============multithreading========  In a single pgm tiny tiny pgm are running.  treads -code which is running independently  --------------------------------------------------  Thr are 2 ways to create a threads  By using thread class  By using Runnable Interface  -----------------------------------------------------------  ====================single threading=======  class Demo  {    }  class Executor  {  public static void main(String args[])  {  //main thread started. //single thread pgm  }  }  //In this eg execution starts from Main Thread and also end at main Thread.  ======================================================================  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  1.creating thread By using thread class  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  1)  class MyThread extends Thread //This class represent Thread  {  public MyThread(String threadName//to give name to the thread) //constructor  {  super(threadName); //super to set the thread name  }  public void run() //the heart of thread is run() & its compulsory.  { //whenever any thread starts,the control goes to run().  for(int i = 1;i<=10;i++) //1000  {  System.out.println(i);  }  }  }  class ThreadDemo  {  public static void main(String args[])  {  MyThread thread1 = new MyThread("First thread"); //to create object of thread class-MyThread here  MyThread thread= new MyThread("Second thread");  //all thread ca2 = new MyThread("Second thread");  //all thread can run simulteneousle  thread1.start(); //It will execute run method for thread1  thread2.start(); //It will execute run method for thread2  //start () to start(run) thread which is in thread class.  //thread is also an object  }  o/p-  1  2  3  4  5  6  7  8  9  10  1  2  3  4  5  6  7  8  9  10  //o/p is showing in sequential manner but both treads are running simultweneously  //if u check with large number u can see the change.  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  2)To display main thread  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  class MyThread extends Thread  {  public MyThread(String threadName)  {  super(threadName);  }  public void run()  {  for(int i = 1;i<=10;i++)  {  System.out.println(i);  }  }  }  class ThreadDemo  {  public static void main(String args[])  {  System.out.println("Main Thread Started");  MyThread thread1 = new MyThread("First thread");  MyThread thread2 = new MyThread("Second thread");    thread1.start();  thread2.start();  System.out.println("Main Thread ended");  }  }  o/p  Main Thread Started  Main Thread ended  1  2  3  4  5  6  7  8  9  10  1  2  3  4  5  6  7  8  9  10  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  In the above pgm main thread start and end before the other thread to overcome we have done below changes.  we use sleep().  3)  class MyThread extends Thread  {  public MyThread(String threadName) //thread1 & thread2 execute here  {  super(threadName);  }  public void run()  {  for(int i = 1;i<=10;i++)  {  System.out.println(i);  }  }  }  class ThreadDemo  {  public static void main(String args[])  {  System.out.println("Main Thread Started");  MyThread thread1 = new MyThread("First thread");  MyThread thread2 = new MyThread("Second thread");  thread1.start();  thread2.start();    Thread.sleep(5000); //As we use sleep() where our main thread will sleep //1000milisec=1sec    System.out.println("Main Thread ended");  }  }  o/p====  Exception in thread "main" java.lang.RuntimeException:  Uncompilable source code - unreported exception java.lang.InterruptedException;  must be caught or declared to be thrown  Every time sleep() throws Exception we have to use try catch every time for sleep()  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  ============handling above excp==================  class MyThread extends Thread  {  public MyThread(String threadName) //thread1 & thread2 execute here  {  super(threadName);  }  public void run()  {  for(int i = 1;i<=10;i++)  {  System.out.println(i);  }  }  }  class ThreadDemo  {  public static void main(String args[])  {  System.out.println("Main thread started");    MyThread thread1 = new MyThread("First thread");  MyThread thread2 = new MyThread("Second thread");    thread1.start(); //It will execute run method for thread1  thread2.start(); //It will execute run method for thread2  try  {  Thread.sleep(2000); //1000milisecond = 1 second  }  catch(InterruptedException e)  {  System.out.println(e);  }  System.out.println("Main thread ended");    }  }  o/p=====  Main thread started  1  2  3  4  5  6  7  8  9  10  1  2  3  4  5  6  7  8  9  10  Main thread ended  //here main thread is sleep for 2 sec till that time thread 1 & thread 2 ends thr work  ==========================================================================================  above eg for sleep() is in main()  now we check sleep() for another thread  class MyThread extends Thread  {  public MyThread(String threadName) //thread1 & thread2 execute here  {  super(threadName);  }  public void run()  {  try{  for(int i = 1;i<=10;i++)  {  Thread.sleep(200);  System.out.println(Thread.currentThread().getName()+" : "+i);  }  }  catch(InterruptedException e) //this is prent class of thread so we cant throws excp  {  System.out.println(e);  }  }  }  class ThreadDemo  {  public static void main(String args[])  {  System.out.println("Main thread started");    MyThread thread1 = new MyThread("First thread");  MyThread thread2 = new MyThread("Second thread");    thread1.start(); //It will execute run method for thread1  thread2.start(); //It will execute run method for thread2  try  {  Thread.sleep(4000); //1000milisecond = 1 second  }  catch(InterruptedException e)  {  System.out.println(e);  }  System.out.println("Main thread ended");    }  }  o/p====  Main thread started  First thread : 1  Second thread : 1  Second thread : 2  First thread : 2  Second thread : 3  First thread : 3  Second thread : 4  First thread : 4  First thread : 5  Second thread : 5  First thread : 6  Second thread : 6  Second thread : 7  First thread : 7  First thread : 8  Second thread : 8  First thread : 9  Second thread : 9  First thread : 10  Second thread : 10  //while printing thread1 & thrad@ taking time 2 sec to print on screen.  =============================================================================  --------- To check which is the current thread running---------------  Thread.currentThread();  t.getName()  or  Thread.currentThread().getName()  =======================================================================  class MyThread extends Thread  {  public MyThread(String threadName) //thread1 & thread2 execute here  {  super(threadName);  }  public void run()  {  Thread t = Thread.currentThread();  for(int i = 1;i<=10;i++)  {  System.out.println(Thread.currentThread().getName()+" : "+i);  }  }  }  class ThreadDemo  {  public static void main(String args[])  {  System.out.println("Main thread started");    MyThread thread1 = new MyThread("First thread");  MyThread thread2 = new MyThread("Second thread");    thread1.start(); //It will execute run method for thread1  thread2.start(); //It will execute run method for thread2  try  {  Thread.sleep(2000); //1000milisecond = 1 second  }  catch(InterruptedException e)  {  System.out.println(e);  }  System.out.println("Main thread ended");    }  }  o/p=  Second thread : 1  Second thread : 2  Second thread : 3  Second thread : 4  Second thread : 5  Second thread : 6  Second thread : 7  Second thread : 8  Second thread : 9  Second thread : 10  First thread : 1  First thread : 2  First thread : 3  First thread : 4  First thread : 5  First thread : 6  First thread : 7  First thread : 8  First thread : 9  First thread : 10  Main thread ended  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  =======================================================================================  2.By using Runnable Interface  ======================================================================================-  class MyThread implements Runnable  {  Thread t; //t - object of thread    public MyThread(String threadName)  {  t = new Thread(this,threadName); //Wrapping of current objS  }  public void run()  {  try {  for(int i = 1; i <= 10; i++)  {  Thread.sleep(200);  System.out.println(Thread.currentThread().getName() + " : " +i);  }  }  catch (InterruptedException e)  {  System.out.println(e);  }  }    }  class ThreadDemo  {  public static void main(String args[])  {  //System.out.println(Thread.currentThread().getName()+" started"); // //1.to check name of current thread-main  Thread mt = Thread.currentThread(); // gives current thread reference  mt.setName("Main thread");  System.out.println(mt.getName()+" started"); // Main thread started  MyThread thread1 = new MyThread("First Thread");  MyThread thread2 = new MyThread("Second Thread");    thread1.t.start(); // execute run() for thread1  thread2.t.start(); // execute run() for thread2    try{  Thread.sleep(4000); //1000 millisecond = 1 second  }  catch (InterruptedException e)  {  System.out.println(e);  }  System.out.println("Main thread ended");  }  }  o/p=  Main thread started  Second Thread : 1  First Thread : 1  First Thread : 2  Second Thread : 2  First Thread : 3  Second Thread : 3  First Thread : 4  Second Thread : 4  Second Thread : 5  First Thread : 5  Second Thread : 6  First Thread : 6  First Thread : 7  Second Thread : 7  First Thread : 8  Second Thread : 8  Second Thread : 9  First Thread : 9  First Thread : 10  Second Thread : 10  Main thread ended  if only want to use run()of thread class.  ===========================================================================  to give the priority  class MyThread extends Thread  {  public MyThread(String threadName)  {  super(threadName);  }  public void run()  {  try {  for(int i = 1; i <= 10; i++)  {  Thread.sleep(200);  System.out.println(Thread.currentThread().getName() + " : " +i);  }  }  catch (InterruptedException e)  {  System.out.println(e);  }  }  public void fun()  {  }  }  class ThreadDemo  {  public static void main(String args[])  {  System.out.println("Main thread started");  MyThread thread1 = new MyThread("First Thread");  MyThread thread2 = new MyThread("Second Thread");    thread1.start(); // execute run() for thread1  thread2.start(); // execute run() for thread2  thread1.setPriority(Thread.MIN\_PRIORITY+2); //priority : 3=5-2 1  thread2.setPriority(Thread.NORM\_PRIORITY+2); //priority : 7=5+2 10    try{  thread1.join();  thread2.join();  }  catch (InterruptedException e)  {  System.out.println(e);  }  System.out.println("Main thread ended");  }  }  o/p====  Main thread started  Second Thread : 1  First Thread : 1  First Thread : 2  Second Thread : 2  First Thread : 3  Second Thread : 3  Second Thread : 4  First Thread : 4  First Thread : 5  Second Thread : 5  Second Thread : 6  First Thread : 6  First Thread : 7  Second Thread : 7  First Thread : 8  Second Thread : 8  Second Thread : 9  First Thread : 9  Second Thread : 10  First Thread : 10  Main thread ended  \*/ |

**Multithreading in**[**Java**](https://www.javatpoint.com/java-tutorial) is a process of executing multiple threads simultaneously.

A thread is a lightweight sub-process, the smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

However, we use multithreading than multiprocessing because threads use a shared memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.

Java Multithreading is mostly used in games, animation, etc.

### Advantages of Java Multithreading

1) It **doesn't block the user** because threads are independent and you can perform multiple operations at the same time.

2) You **can perform many operations together, so it saves time**.

3) Threads are **independent**, so it doesn't affect other threads if an exception occurs in a single thread.

## Multitasking

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved in two ways:

* **Process-based Multitasking (Multiprocessing)**
* **Thread-based Multitasking (Multithreading)**

### 1) Process-based Multitasking (Multiprocessing)

* Each process has an address in memory. In other words, each process allocates a separate memory area.
* A process is heavyweight.
* Cost of communication between the process is high.
* Switching from one process to another requires some time for saving and loading [registers](https://www.javatpoint.com/register-memory), memory maps, updating lists, etc.

### 2) Thread-based Multitasking (Multithreading)

* Threads share the same address space.
* A thread is lightweight.
* Cost of communication between the thread is low.

#### Note: At least one process is required for each thread.

## What is Thread in java

A thread is a lightweight subprocess, the smallest unit of processing. It is a separate path of execution.

Threads are independent. If there occurs exception in one thread, it doesn't affect other threads. It uses a shared memory area.



As shown in the above figure, a thread is executed inside the process. There is context-switching between the threads. There can be multiple processes inside the [OS](https://www.javatpoint.com/os-tutorial), **and one process can have multiple threads.**

#### Note: At a time one thread is executed only.

## Java Thread class

Java provides **Thread class** to achieve thread programming. Thread class provides [constructors](https://www.javatpoint.com/java-constructor) and methods to create and perform operations on a thread. Thread class extends [Object class](https://www.javatpoint.com/object-class) and implements Runnable interface.

## Java Thread Methods

|  |  |  |  |
| --- | --- | --- | --- |
| **S.N.** | **Modifier and Type** | **Method** | **Description** |
| 1) | void | [start()](https://www.javatpoint.com/java-thread-start-method) | It is used to start the execution of the thread. |
| 2) | void | [run()](https://www.javatpoint.com/java-thread-run-method) | It is used to do an action for a thread. |
| 3) | static void | [sleep()](https://www.javatpoint.com/java-thread-sleep-method) | It sleeps a thread for the specified amount of time. |
| 4) | static Thread | [currentThread()](https://www.javatpoint.com/java-thread-currentthread-method) | It returns a reference to the currently executing thread object. |
| 5) | void | [join()](https://www.javatpoint.com/java-thread-join-method) | It waits for a thread to die. |
| 6) | int | [getPriority()](https://www.javatpoint.com/java-thread-getpriority-method) | It returns the priority of the thread. |
| 7) | void | [setPriority()](https://www.javatpoint.com/java-thread-setpriority-method) | It changes the priority of the thread. |
| 8) | String | [getName()](https://www.javatpoint.com/java-thread-getname-method) | It returns the name of the thread. |
| 9) | void | [setName()](https://www.javatpoint.com/java-thread-setname-method) | It changes the name of the thread. |
| 10) | long | [getId()](https://www.javatpoint.com/java-thread-getid-method) | It returns the id of the thread. |
| 11) | boolean | [isAlive()](https://www.javatpoint.com/java-thread-isalive-method) | It tests if the thread is alive. |
| 12) | static void | [yield()](https://www.javatpoint.com/java-thread-yield-method) | It causes the currently executing thread object to pause and allow other threads to execute temporarily. |
| 13) | void | [suspend()](https://www.javatpoint.com/java-thread-suspend-method) | It is used to suspend the thread. |
| 14) | void | [resume()](https://www.javatpoint.com/java-thread-resume-method) | It is used to resume the suspended thread. |
| 15) | void | [stop()](https://www.javatpoint.com/java-thread-stop-method) | It is used to stop the thread. |
| 16) | void | [destroy()](https://www.javatpoint.com/java-thread-destroy-method) | It is used to destroy the thread group and all of its subgroups. |
| 17) | boolean | [isDaemon()](https://www.javatpoint.com/java-thread-isdaemon-method) | It tests if the thread is a daemon thread. |
| 18) | void | [setDaemon()](https://www.javatpoint.com/java-thread-setdaemon-method) | It marks the thread as daemon or user thread. |
| 19) | void | [interrupt()](https://www.javatpoint.com/java-thread-interrupt-method) | It interrupts the thread. |
| 20) | boolean | [isinterrupted()](https://www.javatpoint.com/java-thread-isinterrupted-method) | It tests whether the thread has been interrupted. |
| 21) | static boolean | [interrupted()](https://www.javatpoint.com/java-thread-interrupted-method) | It tests whether the current thread has been interrupted. |
| 22) | static int | [activeCount()](https://www.javatpoint.com/java-thread-activecount-method) | It returns the number of active threads in the current thread's thread group. |
| 23) | void | [checkAccess()](https://www.javatpoint.com/java-thread-checkaccess-method) | It determines if the currently running thread has permission to modify the thread. |
| 24) | static boolean | [holdLock()](https://www.javatpoint.com/java-thread-holdlock-method) | It returns true if and only if the current thread holds the monitor lock on the specified object. |
| 25) | static void | [dumpStack()](https://www.javatpoint.com/java-thread-dumpstack-method) | It is used to print a stack trace of the current thread to the standard error stream. |

# Life cycle of a Thread (Thread States)

A thread can be in one of the five states. According to sun, there is only 4 states in **thread life cycle in java** new, runnable, non-runnable and terminated. There is no running state.

But for better understanding the threads, we are explaining it in the 5 states.

The life cycle of the thread in java is controlled by JVM. The java thread states are as follows:

1. New
2. Runnable
3. Running
4. Non-Runnable (Blocked)
5. Terminated



|  |
| --- |
| 1) New  The thread is in new state if you create an instance of Thread class but before the invocation of start() method. |

2) Runnable

The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.

3) Running

The thread is in running state if the thread scheduler has selected it.

4) Non-Runnable (Blocked)

This is the state when the thread is still alive, but is currently not eligible to run.

5) Terminated

A thread is in terminated or dead state when its run() method exits.

|  |  |  |  |
| --- | --- | --- | --- |
| **How to create thread**  There are two ways to create a thread:   1. By extending Thread class 2. By implementing Runnable interface.  Thread class:  |  | | --- | | Thread class provide constructors and methods to create and perform operations on a thread.Thread class extends Object class and implements Runnable interface. |  Commonly used Constructors of Thread class:  |  | | --- | | * Thread() * Thread(String name) * Thread(Runnable r) * Thread(Runnable r,String name) |  Commonly used methods of Thread class:  |  | | --- | | 1. **public void run():**is used to perform action for a thread. 2. **public void start():**starts the execution of the thread.JVM calls the run() method on the thread. 3. **public void sleep(long miliseconds):**Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds. 4. **public void join():**waits for a thread to die. 5. **public void join(long miliseconds):**waits for a thread to die for the specified miliseconds. 6. **public int getPriority():**returns the priority of the thread. 7. **public int setPriority(int priority):**changes the priority of the thread. 8. **public String getName():**returns the name of the thread. 9. **public void setName(String name):**changes the name of the thread. 10. **public Thread currentThread():**returns the reference of currently executing thread. 11. **public int getId():**returns the id of the thread. 12. **public Thread.State getState():**returns the state of the thread. 13. **public boolean isAlive():**tests if the thread is alive. 14. **public void yield():**causes the currently executing thread object to temporarily pause and allow other threads to execute. 15. **public void suspend():**is used to suspend the thread(depricated). 16. **public void resume():**is used to resume the suspended thread(depricated). 17. **public void stop():**is used to stop the thread(depricated). 18. **public boolean isDaemon():**tests if the thread is a daemon thread. 19. **public void setDaemon(boolean b):**marks the thread as daemon or user thread. 20. **public void interrupt():**interrupts the thread. 21. **public boolean isInterrupted():**tests if the thread has been interrupted. 22. **public static boolean interrupted():**tests if the current thread has been interrupted. | |

### Runnable interface:

|  |
| --- |
| The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interface have only one method named run(). |

|  |
| --- |
| 1. **public void run():**is used to perform action for a thread. |

### Starting a thread:

|  |
| --- |
| **start() method** of Thread class is used to start a newly created thread. It performs following tasks:   * A new thread starts(with new callstack). * The thread moves from New state to the Runnable state. * When the thread gets a chance to execute, its target run() method will run. |

### 1) Java Thread Example by extending Thread class

1. **class** Multi **extends** Thread{
2. **public** **void** run(){
3. System.out.println("thread is running...");
4. }
5. **public** **static** **void** main(String args[]){
6. Multi t1=**new** Multi();
7. t1.start();
8. }
9. }

Output:thread is running...

### 2) Java Thread Example by implementing Runnable interface

1. **class** Multi3 **implements** Runnable{
2. **public** **void** run(){
3. System.out.println("thread is running...");
4. }
6. **public** **static** **void** main(String args[]){
7. Multi3 m1=**new** Multi3();
8. Thread t1 =**new** Thread(m1);
9. t1.start();
10. }
11. }

Output:thread is running...

|  |
| --- |
| If you are not extending the Thread class,your class object would not be treated as a thread object.So you need to explicitely create Thread class object.We are passing the object of your class that implements Runnable so that your class run() method may execute. |

# Thread Scheduler in Java

**Thread scheduler** in java is the part of the JVM that decides which thread should run.

There is no guarantee that which runnable thread will be chosen to run by the thread scheduler.

Only one thread at a time can run in a single process.

# Thread Scheduler in Java

**Thread scheduler** in java is the part of the JVM that decides which thread should run.

There is no guarantee that which runnable thread will be chosen to run by the thread scheduler.

Only one thread at a time can run in a single process.

The thread scheduler mainly uses preemptive or time slicing scheduling to schedule the threads.

### Difference between preemptive scheduling and time slicing

* Under preemptive scheduling, the highest priority task executes until it enters the waiting or dead states or a higher priority task comes into existence.
* Under time slicing, a task executes for a predefined slice of time and then reenters the pool of ready tasks. The scheduler then determines which task should execute next, based on priority and other factors.

# Sleep method in java

The sleep() method of Thread class is used to sleep a thread for the specified amount of time.

## Syntax of sleep() method in java

The Thread class provides two methods for sleeping a thread:

* public static void sleep(long miliseconds)throws InterruptedException
* public static void sleep(long miliseconds, int nanos)throws InterruptedException

Example of sleep method in java

1. **class** TestSleepMethod1 **extends** Thread{
2. **public** **void** run(){
3. **for**(**int** i=1;i<5;i++){
4. **try**{Thread.sleep(500);}**catch**(InterruptedException e){System.out.println(e);}
5. System.out.println(i);
6. }
7. }
8. **public** **static** **void** main(String args[]){
9. TestSleepMethod1 t1=**new** TestSleepMethod1();
10. TestSleepMethod1 t2=**new** TestSleepMethod1();
12. t1.start();
13. t2.start();
14. }
15. }

Output:

1

1

2

2

3

3

4

4

As you know well that at a time only one thread is executed. If you sleep a thread for the specified time,the thread shedular picks up another thread and so on.

# Can we start a thread twice

No. After starting a thread, it can never be started again. If you does so, an IllegalThreadStateException is thrown. In such case, thread will run once but for second time, it will throw exception.

Let's understand it by the example given below:

1. **public** **class** TestThreadTwice1 **extends** Thread{
2. **public** **void** run(){
3. System.out.println("running...");
4. }
5. **public** **static** **void** main(String args[]){
6. TestThreadTwice1 t1=**new** TestThreadTwice1();
7. t1.start();
8. t1.start();
9. }
10. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestThreadTwice1)

running

Exception in thread "main" java.lang.IllegalThreadStateException

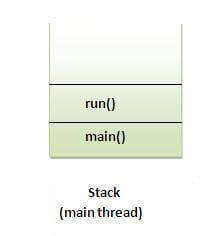
# What if we call run() method directly instead start() method?

|  |
| --- |
| * Each thread starts in a separate call stack. * Invoking the run() method from main thread, the run() method goes onto the current call stack rather than at the beginning of a new call stack. |

1. **class** TestCallRun1 **extends** Thread{
2. **public** **void** run(){
3. System.out.println("running...");
4. }
5. **public** **static** **void** main(String args[]){
6. TestCallRun1 t1=**new** TestCallRun1();
7. t1.run();//fine, but does not start a separate call stack
8. }
9. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestCallRun1)

Output:running...

 ***Problem if you direct call run() method***

1. **class** TestCallRun2 **extends** Thread{
2. **public** **void** run(){
3. **for**(**int** i=1;i<5;i++){
4. **try**{Thread.sleep(500);}**catch**(InterruptedException e){System.out.println(e);}
5. System.out.println(i);
6. }
7. }
8. **public** **static** **void** main(String args[]){
9. TestCallRun2 t1=**new** TestCallRun2();
10. TestCallRun2 t2=**new** TestCallRun2();
12. t1.run();
13. t2.run();
14. }
15. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestCallRun2)

Output:1

2

3

4

5

1

2

3

4

5

|  |
| --- |
| As you can see in the above program that there is no context-switching because here t1 and t2 will be treated as normal object not thread object. |

# The join() method

The join() method waits for a thread to die. In other words, it causes the currently running threads to stop executing until the thread it joins with completes its task.

### Syntax:

|  |
| --- |
| public void join()throws InterruptedException |
| public void join(long milliseconds)throws InterruptedException |

***Example of join() method***

1. **class** TestJoinMethod1 **extends** Thread{
2. **public** **void** run(){
3. **for**(**int** i=1;i<=5;i++){
4. **try**{
5. Thread.sleep(500);
6. }**catch**(Exception e){System.out.println(e);}
7. System.out.println(i);
8. }
9. }
10. **public** **static** **void** main(String args[]){
11. TestJoinMethod1 t1=**new** TestJoinMethod1();
12. TestJoinMethod1 t2=**new** TestJoinMethod1();
13. TestJoinMethod1 t3=**new** TestJoinMethod1();
14. t1.start();
15. **try**{
16. t1.join();
17. }**catch**(Exception e){System.out.println(e);}
19. t2.start();
20. t3.start();
21. }
22. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestJoinMethod1)

Output:1

2

3

4

5

1

1

2

2

3

3

4

4

5

5

|  |
| --- |
| As you can see in the above example,when t1 completes its task then t2 and t3 starts executing. |

***Example of join(long miliseconds) method***

1. **class** TestJoinMethod2 **extends** Thread{
2. **public** **void** run(){
3. **for**(**int** i=1;i<=5;i++){
4. **try**{
5. Thread.sleep(500);
6. }**catch**(Exception e){System.out.println(e);}
7. System.out.println(i);
8. }
9. }
10. **public** **static** **void** main(String args[]){
11. TestJoinMethod2 t1=**new** TestJoinMethod2();
12. TestJoinMethod2 t2=**new** TestJoinMethod2();
13. TestJoinMethod2 t3=**new** TestJoinMethod2();
14. t1.start();
15. **try**{
16. t1.join(1500);
17. }**catch**(Exception e){System.out.println(e);}
19. t2.start();
20. t3.start();
21. }
22. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestJoinMethod2)

Output:1

2

3

1

4

1

2

5

2

3

3

4

4

5

5

|  |
| --- |
| In the above example,when t1 is completes its task for 1500 miliseconds(3 times) then t2 and t3 starts executing. |

### getName(),setName(String) and getId() method:

|  |
| --- |
| public String getName() |
| public void setName(String name) |
| public long getId() |

1. **class** TestJoinMethod3 **extends** Thread{
2. **public** **void** run(){
3. System.out.println("running...");
4. }
5. **public** **static** **void** main(String args[]){
6. TestJoinMethod3 t1=**new** TestJoinMethod3();
7. TestJoinMethod3 t2=**new** TestJoinMethod3();
8. System.out.println("Name of t1:"+t1.getName());
9. System.out.println("Name of t2:"+t2.getName());
10. System.out.println("id of t1:"+t1.getId());
12. t1.start();
13. t2.start();
15. t1.setName("Sonoo Jaiswal");
16. System.out.println("After changing name of t1:"+t1.getName());
17. }
18. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestJoinMethod3)

Output:Name of t1:Thread-0

Name of t2:Thread-1

id of t1:8

running...

After changling name of t1:Sonoo Jaiswal

running...

### The currentThread() method:

|  |
| --- |
| The currentThread() method returns a reference to the currently executing thread object. |

### Syntax:

|  |
| --- |
| public static Thread currentThread() |

***Example of currentThread() method***

1. **class** TestJoinMethod4 **extends** Thread{
2. **public** **void** run(){
3. System.out.println(Thread.currentThread().getName());
4. }
5. }
6. **public** **static** **void** main(String args[]){
7. TestJoinMethod4 t1=**new** TestJoinMethod4();
8. TestJoinMethod4 t2=**new** TestJoinMethod4();
10. t1.start();
11. t2.start();
12. }
13. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestJoinMethod4)

Output:Thread-0

Thread-1

# Naming Thread and Current Thread

## Naming Thread

The Thread class provides methods to change and get the name of a thread. By default, each thread has a name i.e. thread-0, thread-1 and so on. By we can change the name of the thread by using setName() method. The syntax of setName() and getName() methods are given below:

1. **public String getName():** is used to return the name of a thread.
2. **public void setName(String name):** is used to change the name of a thread.

## Example of naming a thread

1. **class** TestMultiNaming1 **extends** Thread{
2. **public** **void** run(){
3. System.out.println("running...");
4. }
5. **public** **static** **void** main(String args[]){
6. TestMultiNaming1 t1=**new** TestMultiNaming1();
7. TestMultiNaming1 t2=**new** TestMultiNaming1();
8. System.out.println("Name of t1:"+t1.getName());
9. System.out.println("Name of t2:"+t2.getName());
11. t1.start();
12. t2.start();
14. t1.setName("Sonoo Jaiswal");
15. System.out.println("After changing name of t1:"+t1.getName());
16. }
17. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestMultiNaming1)

Output:Name of t1:Thread-0

Name of t2:Thread-1

id of t1:8

running...

After changeling name of t1:Sonoo Jaiswal

running...

## Current Thread

The currentThread() method returns a reference of currently executing thread.

1. **public** **static** Thread currentThread()

### Example of currentThread() method

1. **class** TestMultiNaming2 **extends** Thread{
2. **public** **void** run(){
3. System.out.println(Thread.currentThread().getName());
4. }
5. **public** **static** **void** main(String args[]){
6. TestMultiNaming2 t1=**new** TestMultiNaming2();
7. TestMultiNaming2 t2=**new** TestMultiNaming2();
9. t1.start();
10. t2.start();
11. }
12. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestMultiNaming2)

Output:Thread-0

Thread-1