# What is the Predicate in Java 8?

In Java 8 Predicate interface is defined in the java.util.function package. It contains one [abstract method](https://javagoal.com/abstract-method-in-java/) that is called test() method. This interface provides functionality to test/evaluate any condition.

This interface is used in a [lambda expression](https://javagoal.com/java-8-lambda-expressions/) and [method reference](https://javagoal.com/method-reference-in-java-8/) to evaluate the condition.

@FunctionalInterface

public interface Predicate<T>

{

// Body of Interface

}

Where T, The type of input of Predicate. Basically, it is a data type for eg: String, Integer.  
boolean test(T t) method: This method is used to test the given input with the predicate. Its return type is boolean.

It returns true if given argument matched with predicate otherwise return false.

boolean test(T t);

This interface contains a few extra methods but since they all come with a default implementation, you do not have to implement these extra methods.

**How to use** **the Predicate interface in java 8?**

First of all, we will see how to use the Java 8**Predicate interface** without a [**lambda expression**.](https://javagoal.com/java-8-lambda-expressions/)

import java.util.function.Predicate;

public **class** ExampleOfPredicateWithoutLambda

{

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

{

String stringOne = "Hello";

Predicate<String> predicate = **new** Predicate<String>() {

@Override

public boolean test(String stringTwo)

{

**return** stringOne.equals(stringTwo);}};

System.out.println("It test the given string with predicate: ");

System.out.println(predicate.test("Hi"));

System.out.println(predicate.test("Hello"));

}}

***Output:****It test the given string with predicate: false  
true*

Now, we will see how you can use the **Predicate interface** with a [**lambda expression**](https://javagoal.com/java-8-lambda-expressions/)**:**

**import** *java.util.function.Predicate*;

**public** **class** ExampleOfPredicateWithLambda

{

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

{

// Creating predicate by use of lambda expression

Predicate<**String**> predicate = (stringOne) -> (stringOne.equals("Hello"));

System.out.println("Is these Strings are equal = "+predicate.test("Hi"));

System.out.println("Is these Strings are equal = "+predicate.test("Hello"));}}

***Output:****Is these Strings are equal = false  
Is these Strings are equal = true*

**Let’s discuss Predicate with user-defined class**

**public** **class** ExampleOfPredicateForStudent

{

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

{

// Adding data of Student

Student studentOne = new Student(1, "MCA", "Ram");

Student studentTwo = new Student(2, "MSC", "Sham");

Predicate<Student> predicate = (student) -> (student.getRollNo() == 2);

System.out.println("Is student rollNo is 2 = "+predicate.test(studentOne));

System.out.println("Is student rollNo is 2 = "+predicate.test(studentTwo));

}}

***Output:****Is student rollNo is 2 = false  
Is student rollNo is 2 = true*

import java.util.ArrayList;

import java.util.List;

import java.util.function.Predicate;

import java.util.stream.Collectors;

public **class** ExampleOfPredicateWithCollection

{

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

{

Book bookOne = **new** Book(1, "ABC", "shiva");

Book bookTwo = **new** Book(5, "DEF", "RAMA");

Book bookThree = **new** Book(8, "GHI", "Krishna");

List<Book> books = **new** ArrayList<Book>();

books.add(bookOne);

books.add(bookTwo);

books.add(bookThree);

Predicate<Book> predicate = (book) -> (book.getBookId() >= 5);

books = books.stream().filter(predicate).collect(Collectors.<Book>toList());

**for**(Book book : books)

System.out.println("BookId = "+ book.getBookId());

} }

***Output:****BookId = 5  
BookId = 8*

You see in the above example, we created Predicate and used it in a filter.  The filter method of **Stream** takes **Predicate** as an argument.

### Advantage of Predicate

1. By use of Java Predicate, you can move the conditional code in one place.  
2. It improves code maintenance because if you want to make any change in condition then you have to make it in a central place.

# **Why predicate in java 8**

[Predicate interface](https://javagoal.com/java-8-predicate/) evaluates the condition and returns the result in boolean form. In the [Predicate interface](https://javagoal.com/java-8-predicate/), we have an abstract method test(T t). Whenever you are using the [Predicate interface](https://javagoal.com/java-8-predicate/) then you must have to provide the body to **test(T t) method**. You can implement it by using [lambda expression](https://javagoal.com/java-8-lambda-expressions/) or by use **implements** keyword.

By using a [Predicate interface](https://javagoal.com/java-8-predicate/) the [lambda expression](https://javagoal.com/java-8-lambda-expressions/) provides optimized code. In Java 8, [Stream](https://javagoal.com/java-8-stream/)has some methods that expect predicate as input so that they can produce output. Let’s see the**java predicate example**

Stream<T> filter(Predicate<? super T> predicate);

It accepts the predicate as input and filters the objects according to the condition defined in the predicate.

## ****Use of Predicate in Real-life example****

Let’s say you have a record of colleges and want to print the name of the college which has Rank less than 4. So, you must have to check the Rank of each college. So, we will do this task with two approaches.  
The first approach goes without predicate and second with the predicate.

**Here is the first approach without the predicate**

public **class** College

{

String name;

int rank;

String city;

*.List*;

**public** **class** ExampleWithoutPredicate {

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

{

College ClgOne = new College("MLN", 1, "YamunaNagar");

College clgTwo = new College("KUK", 4, "KKR");

College clgThree = new College("JJP", 3, "Delhi");

College clgFour = new College("KKL", 6, "UP");

College clgFive = new College("DPS", 8, "Ambala");

College clgSix = new College("JNU", 2, "J&K");

List<College> colleges = new ArrayList<College>();

colleges.add(ClgOne);colleges.add(clgTwo);colleges.add(clgThree);colleges.add(clgFour);colleges.add(clgFive);

colleges.add(clgSix);

// Filter and print the college have rank less than 4

System.out.println("Filtering and Printing by use of loop");

**for**(College clg : colleges)

{

**if**(clg.getRank() < 4)

{

System.out.println("College Name: "+ clg.getName());

**Output:** Filtering and Printing by use of loop  
College Name: MLN  
College Name: JJP  
College Name: JNU

In the above example, we are filtering and printing the college name by use of If statement.

**Here is the second approach with the predicate**

**public** **class** College

{

**String** name;

**int** rank;

**String** city;

**public** College(**String** name, **int** rank, **String** city)

{

**this**.name = name;

**this**.rank = rank;

**this**.city = city;

}

**public** **String** getName() {

**return** name;

}

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

**this**.name = name;

}

**public** **int** getRank() {

**return** rank;

}

**public** **void** setRank(**int** rank) {

**this**.rank = rank;

}

**public** **String** getCity() {

**return** city;

}

**public** **void** setCity(**String** city) {

**this**.city = city;

}

}

**import** *java.util.ArrayList*;

**import** *java.util.List*;

**public** **class** ExampleWithPredicate

{

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

{

College ClgOne = new College("MLN", 1, "YamunaNagar");

College clgTwo = new College("KUK", 4, "KKR");

College clgThree = new College("JJP", 3, "Delhi");

College clgFour = new College("KKL", 6, "UP");

College clgFive = new College("DPS", 8, "Ambala");

College clgSix = new College("JNU", 2, "J&K");

List<College> colleges = new ArrayList<College>();

colleges.add(ClgOne);

colleges.add(clgTwo);

colleges.add(clgThree);

colleges.add(clgFour);

colleges.add(clgFive);

colleges.add(clgSix);

// Filter the college have rank less than 4

System.out.println("Filtering and Printng by use of loop");

colleges.stream().filter(clg -> (clg.getRank() < 4)).forEach((clg) -> System.out.println("College Name: "+clg.getRank()));

}

}

**Output:** Filtering and Printing by use of loop  
College Name: 1  
College Name: 3  
College Name: 2

In this example filtering and printing the college names by use of Streams. In JAVA, Streams have a filter method that takes Predicate as input and returns the result.

So now you can see we are filtering and printing all the data in one line. So, it makes code more readable and optimized.

# **Consumer in Java 8**

In **Java 8 Consumer interface**is defined in the **java.util.function** package. It contains one [**abstract method**](https://javagoal.com/abstract-method-in-java/) that is known as **accept() method.**This interface accepts a single input argument and doesn’t return the result.

The **Consumer interface** is useful where an object needs to be consumed. It takes an object as input and you can perform the operation on an object without returning any result. This interface is used in a lambda expression and method reference where the user wants to perform the operation without returning the object.

@FunctionalInterface

**public** **interface** Consumer<T>

{

// Body of Interface

}

Where **T,**The type of input of Consumer. Basically, it is thedata type for e.g: String, Integer.

**void accept(T t) method:**This method is used to consume and perform the operation on the object. Its return type is**void.**

**void** **catch**(T t);

This interface contains a few extra methods but since they all come with a default implementation, you do not have to implement these extra methods.

## ****How to use**** ****Java 8**** ****Consumer interface?****

In Java by use of Consumer, you can perform the different operations on objects as per the requirement. Here we have a common example of such an operation is printing. To print any data, you just need the object and you can print it.

First of all, we will see how to use the**Consumer interface** without a [**lambda expression**](https://javagoal.com/java-8-lambda-expressions/):

**import** *java.util.function.Consumer*;

**public** **class** ExampleOfConsumerWithoutLambda

{

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

{

**String** stringOne = "Hello";

Consumer<**String**> consumer = new Consumer<**String**>() {

@Override

**public** **void** accept(**String** string)

{

System.out.println("Accept method of Consumer = "+ string);

}

};

System.out.println("It accept method consumer the data ");

consumer.accept(stringOne);

}

}

***Output:****It accept method consumer the data  
Accept method of Consumer = Hello*

Now, we will see how you can use the Consumer interface with a [**lambda expression**](https://javagoal.com/java-8-lambda-expressions/)**:**

**import** *java.util.function.Consumer*;

**public** **class** ExampleOfConsumerWithLambda

{

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

{

Consumer<**String**> consumer = (value) -> System.out.println("It is a consumer: "+ value);

consumer.accept("Hi!!!!!");

}

}

***Output:****It is a consumer: Hi!!!!!*

**Let’s discuss Consumer with user-defined class:**

public **class** Student

{

int rollNo;

String className;

String name;

public Student(int rollNo, String className, String name)

{

**this**.rollNo = rollNo;

**this**.className = className;

**this**.name = name;

}

public int getRollNo()

{

**return** rollNo;

}

public **void** setRollNo(int rollNo)

{

**this**.rollNo = rollNo;

}

public String getClassName()

{

**return** className;

}

public **void** setClassName(String className)

{

**this**.className = className;

}

public String getName() {

**return** name;

}

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

**this**.name = name;

}

}

**import** *java.util.function.Consumer*;

**public** **class** ExampleOfConsumerForStudent

{

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

{

// Adding data of Student

Student studentOne = new Student(1, "MCA", "Ram");

Student studentTwo = new Student(2, "MSC", "Sham");

Consumer<Student> consumer = (object) ->

{

System.out.println(object.getName());

System.out.println(object.getRollNo());

System.out.println(object.getClassName());

};

System.out.println("First student data:");

consumer.accept(studentOne);

System.out.println("Second student data:");

consumer.accept(studentTwo);

}

}

***Output:****First student data:  
Ram  
1  
MCA  
Second student data:  
Sham  
2  
MSC*

**Let’s discuss** **Consumer with**[**Collection**](https://javagoal.com/java-collection/)**using**[**foreach**](https://javagoal.com/foreach-loop-in-java-8/)**:**

**public** **class** Book

{

**int** bookId;

**String** name;

**String** authorName;

**public** Book(**int** bookId, **String** name, **String** authorName)

{

**this**.bookId = bookId;

**this**.name = name;

**this**.authorName = authorName;

}

**public** **int** getBookId()

{

**return** bookId;

}

**public** **void** setBookId(**int** bookId)

{

**this**.bookId = bookId;

}

**public** **String** getName()

{

**return** name;

}

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

**this**.name = name;

}

**public** **String** getAuthorName() {

**return** authorName;

}

**public** **void** setAuthorName(**String** authorName) {

**this**.authorName = authorName;

}

}

**import** *java.util.ArrayList*;

**import** *java.util.List*;

**import** *java.util.function.Consumer*;

**public** **class** ExampleOfConsumerWithCollection

{

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

{

Book bookOne = new Book(1, "ABC", "shiva");

Book bookTwo = new Book(5, "DEF", "RAMA");

Book bookThree = new Book(8, "GHI", "Krishna");

List<Book> books = new ArrayList<Book>();

books.add(bookOne);

books.add(bookTwo);

books.add(bookThree);

// Creating consumer

Consumer<Book> consumer = (book) -> System.out.println("Book id :"+book.getBookId());

// foreach accepts object of consumer

books.stream().forEach(consumer);

}

}

***Output:****Book id :1  
Book id :5  
Book id :8*

You see in the above example, we created consumer and used foreach to print all data. Because forEach accepts the object of Consumer.

**Advantage of Consumer:**

1. By use of Consumer, you can move the operation at one place.  
2. It improves code maintenance because if you want to make any change in one place.

**public** **class** Book

{

**int** bookId;

**String** name;

**String** authorName;

**public** Book(**int** bookId, **String** name, **String** authorName)

{

**this**.bookId = bookId;

**this**.name = name;

**this**.authorName = authorName;

}

**public** **int** getBookId()

{

**return** bookId;

}

**public** **void** setBookId(**int** bookId)

{

**this**.bookId = bookId;

}

**public** **String** getName()

{

**return** name;

}

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

**this**.name = name;

}

**public** **String** getAuthorName() {

**return** authorName;

}

**public** **void** setAuthorName(**String** authorName) {

**this**.authorName = authorName;

}

}

**import** *java.util.ArrayList*;

**import** *java.util.List*;

**import** *java.util.function.Consumer*;

**public** **class** ExampleOfConsumerWithCollection

{

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

{

Book bookOne = new Book(1, "ABC", "shiva");

Book bookTwo = new Book(5, "DEF", "RAMA");

Book bookThree = new Book(8, "GHI", "Krishna");

List<Book> books = new ArrayList<Book>();

books.add(bookOne);

books.add(bookTwo);

books.add(bookThree);

ExampleOfConsumerWithCollection obj = new ExampleOfConsumerWithCollection();

// foreach accepts object of consumer

books.stream().forEach(obj.getConsumer());

}

**private** Consumer<Book> getConsumer()

{

// Creating consumer

Consumer<Book> consumer = (book) ->

{

System.out.println("Book id :"+book.getBookId());

System.out.println("Book name:"+ book.getName());

System.out.println("Auther name");

};

**return** consumer;

}

}

***Output:****Book id :1  
Book name:ABC  
Auther name  
Book id :5  
Book name:DEF  
Auther name  
Book id :8  
Book name:GHI  
Auther name*

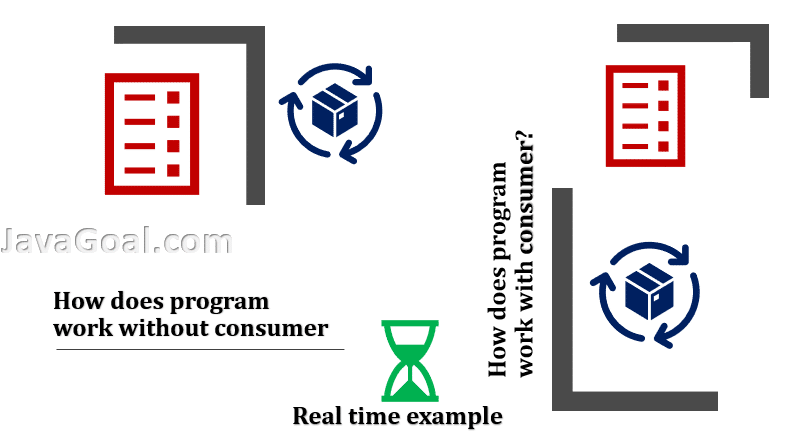
## Why consumer in java 8

As per our recent discussion on [**Consumer**](https://javagoal.com/java-8-consumer-interface/), Now you have some knowledge of the [**Consumer interface**](https://javagoal.com/java-8-consumer-interface/). In this article, we will read Why consumers in java 8 and a real-time java consumer example?

As you already know [**Consumer interface**](https://javagoal.com/java-8-consumer-interface/) consumes the object and doesn’t return the result. In [**Consumer interface**](https://javagoal.com/java-8-consumer-interface/) we have an [**abstract method**](https://javagoal.com/abstract-method-in-java/)**accept(T t)**. Whenever you are using a [**Consumer interface**](https://javagoal.com/java-8-consumer-interface/) then you must have to provide the body to **accept(T t) method**. You can implement it by using lambda expression or by use implements keyword. By using the [**Consumer interface**](https://javagoal.com/java-8-consumer-interface/) the [**lambda expression**](https://javagoal.com/java-8-lambda-expressions/)provides optimized code. You can perform any operation on the object without returning the values. In Java 8, Streams have [**forEach**](https://javagoal.com/foreach-loop-in-java-8/)loop that expects a Consumer as input so that it can perform the operation on it. Let’s see java consumer example:

**void** forEach(Consumer<? **super** T> action);

It accepts the **Consumer**as input, and You can perform the operation on the object.



**Use of Consumer in Real-life example**

Let’s say you have a record of the library and want to print the name of Books in capital letters. So, you must have to perform some operation on the name of each Book. So, we will do this task with two approaches. The first approach goes without Consumer and second with Consumer.

**Here is the first approach without the Consumer**

**public** **class** Book

{

**int** bookId;

**String** name;

**String** authorName;

**public** Book(**int** bookId, **String** name, **String** authorName)

{

**this**.bookId = bookId;

**this**.name = name;

**this**.authorName = authorName;

}

**public** **int** getBookId()

{

**return** bookId;

}

**public** **void** setBookId(**int** bookId)

{

**this**.bookId = bookId;

}

**public** **String** getName()

{

**return** name;

}

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

**this**.name = name;

}

**public** **String** getAuthorName() {

**return** authorName;

}

**public** **void** setAuthorName(**String** authorName) {

**this**.authorName = authorName;

}

}

**import** *java.util.ArrayList*;

**import** *java.util.List*;

**public** **class** ExampleWithoutConsumer {

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

{

Book javaBook = new Book(123, "java begin", "RKK");

Book digitalBook = new Book(432, "Digital logics", "KG");

Book mathematicsBook = new Book(412, "mathmaticsc", "DL saini");

List<Book> books = new ArrayList<Book>();

books.add(javaBook);

books.add(digitalBook);

books.add(mathematicsBook);

// print all the name of books

System.out.println("Printng by use of loop");

**for**(Book book : books)

{

System.out.println("Book Name: "+ book.getName().toUpperCase());

}

}

}

***Output:****Printing by use of loop  
Book Name: JAVA BEGIN  
Book Name: DIGITAL LOGICS  
Book Name: MATHMATICSC*

In the above example, we are printing the name of books and perform one operation on the books name.

**Here is the second approach with the Consumer:**

public **class** Book

{

int bookId;

String name;

String authorName;

public Book(int bookId, String name, String authorName)

{

**this**.bookId = bookId;

**this**.name = name;

**this**.authorName = authorName;

}

public int getBookId()

{

**return** bookId;

}

public **void** setBookId(int bookId)

{

**this**.bookId = bookId;

}

public String getName()

{

**return** name;

}

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

**this**.name = name;

}

public String getAuthorName() {

**return** authorName;

}

public **void** setAuthorName(String authorName) {

**this**.authorName = authorName;

}

}

import java.util.ArrayList;

import java.util.List;

public **class** ExampleWithConsumer

{

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

{

Book javaBook = **new** Book(123, "java begin", "RKK");

Book digitalBook = **new** Book(432, "Digital logics", "KG");

Book mathematicsBook = **new** Book(412, "mathmaticsc", "DL saini");

List<Book> books = **new** ArrayList<Book>();

books.add(javaBook);

books.add(digitalBook);

books.add(mathematicsBook);

// print all the name of books

System.out.println("Printng by use of loop");

books.forEach(book -> System.out.println("Book Name: "+ book.getName().toUpperCase()));

}

}

***Output:****Printing by use of loop  
Book Name: JAVA BEGIN  
Book Name: DIGITAL LOGICS  
Book Name: MATHMATICSC*

In this example printing the book names by use of forEach. In JAVA, Streams have **forEach** method that takes Consumer as input.

So now you can see we are printing all the data in one line. So, it makes code more readable and optimized.

# **Supplier interface in Java**

The **supplier interface**is defined in the **java.util.function** package. It contains one [**abstract method**](https://javagoal.com/abstract-method-in-java/) that is known as **get() method.**This interface returns the result of Type **T**. The supplier interface has only one method that supplies the data, unlike the [**consumer interface**](https://javagoal.com/java-8-consumer-interface/) it doesn’t consume any data. In this post we will see what is **supplier functional interface** in detail and discuss it with example.

The **Supplier interface** is useful where you want to get an object of type **T**. It does not take any object as input but produces a value of type **T.**This interface is used in a[**lambda expression**](https://javagoal.com/java-8-lambda-expressions/) and [**method reference**](https://javagoal.com/method-reference-in-java-8/) where the user wants to return object. This interface is only used to get/supply the data to another function.

@FunctionalInterface

public interface Supplier<T>

{

// Body of Interface

}

Where T, The type of object which you want to get from Supplier.  
**T get( ) method:**This method is used to produce an object of type T. Its return type is**T.**

T get() method

### **How to use the supplier functional interface?**

First of all, we will see how to use the**Predicate interface** without a [**lambda expression**](https://javagoal.com/java-8-lambda-expressions/).

import java.util.function.Supplier;

public **class** ExampleOfSupplierWithoutLambda

{

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

{

Supplier<String> supplier = **new** Supplier<String>()

{

@Override

public String get()

{

**return** "Hello";

}

};

System.out.println(supplier.get());

}

}

***Output:****Hello*

Now, we will see how you can use the **Supplier interface** with a [**lambda expression**](https://javagoal.com/java-8-lambda-expressions/)**:**

import java.util.function.Supplier;

public **class** ExampleOfSupplierWithLambda

{

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

{

Supplier<String> supplier = () -> "Hello";

System.out.println(supplier.get());

}

}

***Output:****Hello*

### **Let’s discuss Supplier with user-defined class**

import java.util.function.Supplier;

public **class** ExampleOfSupplierForStudent

{

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

{

Supplier<Student> supplier = () -> **new** Student(5, "MCA", "Ram");

System.out.println("Student name:"+supplier.get().getName());

}

}

***Output:****Student name:Ram*

# **Function interface in Java 8**

## ****What is the function interface in Java 8?****

The**Function interface**is defined in the **java.util.function** package. It contains one [**abstract method**](https://javagoal.com/abstract-method-in-java/) that is known as **apply() method.**This interface accepts the input and provides output. It takes one parameter as input and produces the output.

This interface is used in a [**lambda expression**](https://javagoal.com/java-8-lambda-expressions/) and [**method reference**](https://javagoal.com/method-reference-in-java-8/) to evaluate the condition.

@FunctionalInterface

**public** **interface** Function<T, R>

{

// Body of Interface

}

Where **T**, The type of input to the function.  
**R**, The type of the result of the function.

**R**apply**(T t) method:**This method is used to perform the operation on the given input and produce the result. Its return type is **R**, where **R**is a data type which specified while using the interface.

R apply(T t)

This interface contains a few extra methods but since they all come with a default implementation, you do not have to implement these extra methods.

This method is very useful whenever you want to perform the operation on the object and want results accordingly. The programmer must have to specify the data type of input and result.  
Here T is used for input data type and R used for the result.

## ****How to use**** ****the Function interface?****

First of all, we will see how to use the**Function interface** without a [**lambda expression**](https://javagoal.com/java-8-lambda-expressions/).

**import** *java.util.function.Function*;

**public** **class** ExampleOfFunctionWithoutLambda

{

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

{

Function<**String**, **String**> function = new Function<**String**, **String**>()

{

@Override

**public** **String** apply(**String** string)

{

**return** string.toUpperCase();

}

};

System.out.println(function.apply("Hi"));

System.out.println(function.apply("Hello"));

}

}

***Output:****HI  
HELLO*

You can perform the action according to requirements. Let’s say create an example for the different data type. In this example, we are providing **Integer** type for input and **String** type for Result.

import java.util.function.Function;

public **class** ExampleOfFunctionWithoutLambda

{

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

{

**Function**<Integer, String> **function** = **new** **Function**<Integer, String>()

{

@Override

public String apply(Integer integer)

{

**return** integer.toString();

}

};

System.out.println(**function**.apply(123));

System.out.println(**function**.apply(55555));

}

}

***Output:****123  
55555*

Now, we will see how you can use the **Function interface** with a [**lambda expression**](https://javagoal.com/java-8-lambda-expressions/)**:**

import java.util.function.Function;

public **class** ExampleOfFunctionWithLambda

{

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

{

**Function**<Integer, String> **function** = (value) -> value.toString();

System.out.println(**function**.apply(123));

System.out.println(**function**.apply(55555));

}

}

***Output:****123  
55555*

## ****Advantage of Function****

1. By use of Function Interface, you can move the operational code in one place.  
2. It improves code maintenance because if you want to perform operations on the object then you have to make it in a central place.

# **Optional class in java 8**

In this article, we will discuss how to use the **Optional class** in java 8? What are the benefits of **optional class**? We will also discuss the methods of **Optional class**.

**Here is the table content of the article will we will cover this topic.**  
[**1. What is the Optional class in Java 8?**](https://javagoal.com/optional-class-in-java/#1)[**2. How to use the Optional class?**](https://javagoal.com/optional-class-in-java/#2)[**3. Methods in the Optional class?**](https://javagoal.com/optional-class-in-java/#3)

## ****What is the Optional class in Java 8?****

In Java 8, a new class was introduced that is an **Optional class**. The **optional class** is [**final class**](https://javagoal.com/final-class-in-java/) that is present in**java.util**package. This class helps the programmer to write better code and avoid the use of **null** checks.

**Let’s understand it in brief:**

Every programmer knows about the **NullPointerException**in Java. Many of us must have encountered it. The **NullPointerException**exception occurs only when the programmer tries to use an object reference that doesn’t point any instance. There is the various reason if the object doesn’t point any instance,  maybe it is not initialized, or it initialized with **null**. It is not an easy job to avoid null without using too many **null**checks.

**The solution is given by Java 8:**

To resolve this problem Java 8, introduce a new class**Optional class**. The **Optional class** provides various methods to check the value of the object. The programmer can specify the alternate value to return. It makes the code in readable form and the programmer doesn’t need to specify the **null**check.

First of all, we will see without the use of the **Optional class**.

public **class** WithoutOptional

{

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

{

Integer[] intArray = **new** Integer[5];

**for**(int i = 0; i < intArray.length; i++)

{

String word = intArray[i].toString();

System.out.print(word);

}

}

}

***Output:****Exception in thread “main” java.lang.NullPointerException at First.WithoutOptional.main(WithoutOptional.java:10)*

Now, we will see how you can use the **Optional class:**

import java.util.Optional;

public **class** WithoutOptional

{

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

{

Integer[] intArray = **new** Integer[5];

**for**(int i = 0; i < intArray.length; i++)

{

Optional<Integer> checkNull = Optional.ofNullable(intArray[i]);

**if**(checkNull.isPresent())

{

String word = intArray[i].toString();

System.out.print(word);

}

**else**

System.out.println("String is NULL");

}

}

}

***Output:****String is NULL  
String is NULL  
String is NULL  
String is NULL  
String is NULL*

## ****How to use**** ****the Optional class?****

public final **class** Optional<T> {

/\*\*

\* Common instance **for** {@code empty()}

}

**Optional**is a [**generic class**](https://javagoal.com/java-generics/) in java, so you must specify the type of data that can it hold. The **Optional class** provides a way, to replace the **null**reference of **T** Type with a non-null value. You can directly check the reference is present or absent. Because the **Optional class** either contains any non-null value or nothing.

import java.util.Optional;

public **class** WithOptional

{

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

{

String s = "";

String s1 = "Hello";

Optional<String> objOfOptional = Optional.of(s);

// Return true if it contain non-null value else false

System.out.println(objOfOptional.isPresent());

Optional<String> objOfOptional1 = Optional.of(s1);

// Return true if it contain non-null value else false

System.out.println(objOfOptional1.isPresent());

}

}

***Output:****true  
true*

## ****Methods in the Optional class****

The **optional class** has various methods that are used to perform the operation on the object of optional. It contains static or non-static methods.

**Firstly, we will discuss some methods that are used to create an object of Optional class:**

**1.** **static <T> Optional<T> empty():**It is used to create an empty optional. It means there is no value present.

import java.util.Optional;

public **class** WithOptional

{

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

{

Optional<String> objOfOptional = Optional.empty();

// Return true if it contain non-null value else false

System.out.println("Is value Present = "+objOfOptional.isPresent());

}

}

***Output:****Is value Present = false*

**2. static** **<T> Optional<T> of(T value):** This method is used to create an object of optional that contains a non-null value. If you try to pass any null value it will throw **NullPointerException**.

import java.util.Optional;

public **class** WithOptional

{

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

{

Optional<String> objOfOptional = Optional.of("Hello");

// Return true if it contain non-null value else false

System.out.println("Is value Present = "+objOfOptional.isPresent());

}

}

***Output:****Is value Present = true*

**3. static <T> Optional<T> ofNullable(T value):**This method is used to create an object of optional that may contain a null value.

import java.util.Optional;

public **class** WithOptional

{

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

{

Optional<String> objOfOptional = Optional.ofNullable("Hello");

// Return true if it contain non-null value else false

System.out.println("Is value Present = "+objOfOptional.isPresent());

Optional<String> objOfOptional1 = Optional.ofNullable(**null**);

// Return true if it contain non-null value else false

System.out.println("Is value Present = "+objOfOptional1.isPresent());

}

}

***Output:****Is value Present = true  
Is value Present = false*

**How to check either the value exists/presented or not?**

**4. boolean isPresent():** This method is used to check whether the optional is contain any value or not. It returns true if the value is present otherwise false.

import java.util.Optional;

public **class** WithOptional

{

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

{

Optional<Integer> objOfOptional = Optional.of(1);

// Return true if it contain non-null value else false

System.out.println("Is value Present = "+objOfOptional.isPresent());

Optional<String> objOfOptional1 = Optional.ofNullable(**null**);

// Return true if it contain non-null value else false

System.out.println("Is value Present = "+objOfOptional1.isPresent());

}

}

***Output:****Is value Present = true  
Is value Present = false*

**5. void** **ifPresent(**[**Consumer**](https://javagoal.com/java-8-consumer-interface/)**<? super T> action)**: This method is used to perform given action on given value if the value is present in **optional**, otherwise it doesn’t perform any operation.

import java.util.Optional;

import java.util.function.Consumer;

public **class** WithOptional

{

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

{

Optional<String> objOfOptional = Optional.of("Hi");

objOfOptional.ifPresent((a) -> System.out.println("The optional contain value = "+a));

Optional<String> objOfOptional1 = Optional.ofNullable(**null**);

objOfOptional1.ifPresent((a) -> System.out.println("The optional contain value"));

}

}

***Output:****The optional contain value = Hi*

**6. void** **ifPresentOrElse (**[**Consumer**](https://javagoal.com/java-8-consumer-interface/)**<? super T> action, Runnable emptyAction):** This method is used to perform given action on given value if the value is present in optional, else it performs the **emptyAction**operation. You can specify the action which you want to perform if value id not present.

import java.util.Optional;

import java.util.function.Consumer;

**class** check **implements** Runnable

{

@Override

public **void** run() {

// TODO Auto-generated method stub

}

}

public **class** WithOptional

{

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

{

Optional<String> objOfOptional = Optional.of("Hi");

Optional<String> objOfOptional1 = Optional.ofNullable(**null**);

Consumer<String> consumer = (a) -> System.out.println("The optional contain value = "+a);

Runnable emptyAction = () -> System.out.println("This is else part");

objOfOptional.ifPresentOrElse(consumer, emptyAction);

objOfOptional1.ifPresentOrElse(consumer, emptyAction);

}

}

***Output:****The optional contain value = Hi  
This is else part*

**7. T get():**This method is used to get the value from optional. It returns the value, If the value is present in this Optional, otherwise throws **NoSuchElementException**.

import java.util.Optional;

public **class** WithOptional

{

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

{

Optional<String> objOfOptional = Optional.of("Hi");

System.out.println(objOfOptional.get());

}

}

***Output:****Hi*

**8. T orElse(T other):** This method is used to get the value from **optional**. It returns the value, If the value is present in this **Optional**, otherwise return the specified value provided as the parameter.

import java.util.Optional;

public **class** WithOptional

{

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

{

Optional<String> obj = Optional.of("Hi");

System.out.println("Value from Optional = "+obj.orElse("Hello"));

Optional<String> obj1 = Optional.ofNullable(**null**);

System.out.println("Value from specifeid parameter = "+obj1.orElse("Hello"));

}

}

***Output:****Value from Optional = Hi  
Value from specifeid parameter = Hello*

**9. Optional<T> filter(Predicate<? super T> predicate):** This method returns an object of **Optional**. If the value in present and matched with given [Predicate](https://javagoal.com/java-8-predicate/). Otherwise, it returns an empty Optional.

import java.util.Optional;

public **class** WithOptional

{

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

{

Optional<String> obj = Optional.of("Hello");

Optional<String> objByFilter = obj.filter(a -> a.equals("Hello"));

System.out.println("Is this empty optional = "+ objByFilter.isPresent());

Optional<String> objByFilter1 = obj.filter(a -> a.equals("Hi"));

System.out.println("Is this empty optional = "+ objByFilter1.isPresent());

}

}

***Output:****Is this empty optional = true  
Is this empty optional = false*

# **What is a lambda expression Java?**

Lambda expressions are new features for **JAVA,** but it is already existing in Some other popular programming languages like Scala.  
Java 8 onwards supports functional programming. Because now you can pass a function as a parameter in another function. It is an anonymous function(A function without a name). Java has been lacking this from the beginning. Let’s discuss it in detail.

### **Syntax of Lambda expression in Java8**

(parameters) -> { statements; }

**Round brackets ():** It contains the parameters. The round brackets are mandatory.  
**Parameters:** The parameter can be empty or non-empty as well. These parameters can use in the statement.

**Arrow token** ->**:** It is mandatory it’s just a symbol that links the parameters and statements.

**Curly braces { }:** These are not mandatory if you have one line of the statement.  
**Statements:** You can place the statement or block of code.

Here we are creating valid syntax with examples:

( ) -> { statements; }

()-> { System.out.println("Hi!!!!!"); }

( ) -> statements;

( ) -> System.out.println("Hi!!!!!");

(parameters) -> { statements; }

(int a, int b) -> { System.out.println(a+b); }

(parameters) -> statements;

(int a, int b) -> System.out.println(a+b);

### **Rules in Lambda expression**

1. Lambda expression can have any number of parameters. The parameters are not mandatory. They can be empty or not empty.
2. The type of parameters should be matched with the type of the parameter of the abstract method.
3. The round brackets are not mandatory if lambda expression has a single parameter. But when there is more than one parameter then it is mandatory.
4. The body of the lambda expressions can contain any number of statements.
5. If the body of lambda expression has a single statement, the curly brackets are not mandatory but when there is more than one statement in the body than these must be enclosed in curly brackets.

### **How to use Java 8 lambda expressions**

Lambda expression works with a [**Functional interface**](https://javagoal.com/function-interface-in-java-8). Basically, lambda expression provides the implementation of a [**Functional interface**](https://javagoal.com/function-interface-in-java-8). It represents the instance of the [**Functional Interface**](https://javagoal.com/function-interface-in-java-8).

@FunctionalInterface

**interface** ExampleOfFunctionInterface

{

**public** **void** firstMethod();

}

To understand the working of Lambda we need to understand existing approaches that used to implement the interface.

**public** **class** FirstApprochToImplement **implements** ExampleOfFunctionInterface

{

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

{

FirstApprochToImplement object = new FirstApprochToImplement();

System.out.println("In this class we are implementing the functinal interface");

object.firstMethod();

}

**public** **void** firstMethod()

{

System.out.println("Function interface can contains only one abstract method");

System.out.println("Providing body to abstract method");

}

}

***Output:****In this class we are implementing the functional interface  
Function interface can contains only one abstract method  
Providing body to abstract method*

Here we are implementing the interface and providing the body to the method. After that, we are creating an object to access the method. If you do not know this approach, please read it from [here](https://javagoal.com/interface-in-java/).

**public** **class** SecondApprochToImplement

{

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

{

ExampleOfFunctionInterface object = new ExampleOfFunctionInterface()

{

@Override

**public** **void** firstMethod()

{

System.out.println("Function interface can contains only one abstract method");

}

};

System.out.println("In this class we are provide body to method during the creation of object");

object.firstMethod();

}

}

***Output:****In this class we are provide body to method during the creation of object  
Function interface can contains only one abstract method*

Here we are not implementing the interface. We are creating the object of Interface so we must have to provide the body to the method. After that, we are accessing the method. If you do not know this approach, please read it from [here](https://javagoal.com/anonymous-inner-class/).

**By use of  Lambda expression user neither need to implement the interface nor need to create the**[**anonymous class**](https://javagoal.com/anonymous-inner-class/)**:**

public **class** ThirdApprochToImplement

{

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

{

ExampleOfFunctionInterface object = () ->

{

System.out.println("Function interface can contains only one abstract method");

};

System.out.println("In this class we are provide body to method by use of Lambda expressions.");

object.firstMethod();

}

}

***Output:****In this class we are provide body to method by use of Lambda expressions.  
Function interface can contains only one abstract method*

Let’s take another example of Lambda expression

**import** *java.util.ArrayList*;

**public** **class** LibraryRecords

{

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

{

ArrayList<Library> listOfBooks = new ArrayList<Library>();

listOfBooks.add(new Library(12, "Java begin", "PK mohan"));

listOfBooks.add(new Library(22, "Logical", "RK"));

listOfBooks.add(new Library(14, "Mathematics", "Rama"));

listOfBooks.add(new Library(18, "Java CORE", "Krishana"));

listOfBooks.add(new Library(54, "Data structutre", "MS"));

listOfBooks.stream()

.filter(book -> book.getBookId() < 20)

.forEach(book ->

{

System.out.println("Book Name = "+book.getBookName());

System.out.println("Book Id = "+book.getBookId());

});

}

}

***Output:****Book Name = Java begin  
Book Id = 12  
Book Name = Mathematics  
Book Id = 14  
Book Name = Java CORE  
Book Id = 18*

# **equals() and hashcode() in java**

Before starting the about the **contract between equals() and hashCode() in Java**. We will recommend you should learn about [**hashCode()**](https://javagoal.com/hashcode-method-in-java/)and [**equals() method**](https://javagoal.com/equals-method-in-java/). You should learn about their **default implementation and how to correctly override them**.

In this article, we will discuss the most important topic about the **equals()**and **hashCode() in Java**. Many programmers want to know **why we should override the hashCode() and equals() method**. What if we are overriding only one method? When we should override them.

Here is the table content of the article will we will cover this topic.

## ****Where equals() and hashCode() method exists?****

Each method is defined in the **java.lang.Object**which is the Superclass in Java. Each class inheriting these methods from the **Object class.**The object class provides these methods for comparing objects. If you are working with a simple class, then these methods will be easy to use or override. But if you are working with several classes in large projects then you must understand the concept of**equals() and hashCode() method**.

Let’s see the default implementation of **hashcode() and equals() method**.

// Default implementation of hashCode() method

public native int hashCode();

// Default implementation of equals() method

public boolean equals(Object obj) {

**return** (**this** == obj);

}

This is the default implementation of these methods provided in **Object class**.

## ****What is equal() method****

The **equals() method** is defined in [***Object class***](https://javagoal.com/object-class-in-java/) which is the super most class in Java. This method is used to compare two objects and it returns the boolean value based on the comparison.  
You can override the **equals() method** as per your functionality. Suppose you want to compare the objects own equality condition. Then it is recommended to override **equals(Object obj)** method.

Let’s take the example from the ***String class.***In the **String class**, **equals() method** is overridden and provided the equality condition accordingly. For more detail, you should read it from [here](https://javagoal.com/equals-method-in-java/).

public boolean equals(Object anObject) {

**if** (**this** == anObject) {

**return** **true**;

}

**if** (anObject instanceof String) {

String aString = (String)anObject;

**if** (coder() == aString.coder()) {

**return** isLatin1() ? StringLatin1.equals(value, aString.value)

: StringUTF16.equals(value, aString.value);

}}**return** **false**;

As you can see in the above code the **String class** overridden the method and comparing two strings. It compares the two given strings based on the content of the string. It returns the boolean value. If both strings are not the same then it returns false otherwise, it returns true.

## ****Why OR When we should Override the equals() method?****

The default implementation of the **equals() method** compares only the references of objects. But sometimes default implementation is not useful, and we want to compare the objects as per our requirements. In these types of situations, it is better to override the **equals() method**.

Let’s discuss it with example. We will create a **user-defined class** and let’s see what will be the result of comparison when we will override the **equals() method** or not?  
Let’s take an example of the default implementation of **equals() method.**

public **class** Employee {

int empId;

String name;

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

{

Employee emp1 = **new** Employee();

emp1.name = "Ravi";

emp1.empId = 1;

Employee emp2 = **new** Employee();

emp2.name = "Ram";

emp2.empId = 2;

System.out.println("Is both employee have same id:");

System.out.println(emp1.equals(emp2));

Employee emp3 = **new** Employee();

emp3.name = "Ram";

emp3.empId = 2;

System.out.println(emp2.equals(emp3));

}}

***Output:*** ***Output:*** Is both employee have same id:  
false  
false

**Reason:** You must saw **emp2**and **emp3**have the same name and id. But **equals() method** saying these do not equal because when we compare **emp2** and **emp3**, it is checked whether both **emp2** and **emp3**refer to the same object or not. The compiler made a call to **equals() method**of Object class and it compares the references of the object.

The **emp2**and **emp3** refer to two different objects, hence the value (emp2 == emp3) is false.  
If we create another reference say **emp4**like following:

***Employee emp4 = emp3;***  
then (emp3 == emp4) will give true.

So, how do we check for equality of values inside the objects?  
We can achieve this task by overriding the **equals() method** of Object class. We can override the **equals() method**in our class to check whether two objects have the same data or not.

public **class** Employee

{

int empId;

String name;

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

{

Employee emp1 = **new** Employee();

emp1.name = "Ravi";

emp1.empId = 1;

Employee emp2 = **new** Employee();

emp2.name = "Ram";

emp2.empId = 2;

System.out.println("Is both employee have same id:");

System.out.println(emp1.equals(emp2));

Employee emp3 = **new** Employee();

emp3.name = "Ram";

emp3.empId = 2;

System.out.println(emp2.equals(emp3));

}

@Override

public boolean equals(Object obj)

{

**return** ((Employee)obj).empId == **this**.empId;

}

}

***Output:***Is both employee have same id:falsetrue

## ****What is hashCode() method****

The **hashCode() method** is defined in [***Object class***](https://javagoal.com/object-class-in-java/) which is the super most class in Java. This method **returns** a hash code value for the object. Let’s have a look at code.

1. It returns an integer value of the object memory address. Some programmer thinks it is the memory address of an object which is not correct. It’s the integer representation of the object’s memory address.  
2. It is a **native method**. By means of the native method, it is written in C/C++ language. As you know, in java we can’t find the memory address of the object so that the **hashCode() method** is written in C/C++ that helps find the memory address.

As you already know every class in java is the child of Object class. It means each class inheriting the **hashCode() method** from [**Object class**](https://javagoal.com/object-class-in-java/).  
1. Whenever a new Object is created, JVM creates a new entry of Object in memory with the corresponding hashCode.  
2. The **hashCode() method** should return a unique value for every object.  
3. If two objects are equal according to [**equals() method**](https://javagoal.com/equals-method-in-java/), then their hash code must be the same.

// If obj1 and obj2 are equals and returning true according to the equals() method

Obj1.equals(obj2);

// Then hashCode() of both objects must be equal.

Obj1.hashCode() == obj2.hashCode()

4. If two objects are unequal according to equals() method, their hash code may or may not be equal.

// If obj1 and obj2 are not equals and returning false according to the equals() method

Obj1.equals(obj2);

// Then hashCode() of both objects may be equal or may be not.

Obj1.hashCode() == obj2.hashCode()

Let’s take the example from the **String class**.  
In the **String class**, the **hashCode() method** is overridden and provided the equality condition accordingly.

public int hashCode()

{

int h = hash;

**if** (h == 0 && value.length > 0) {

hash = h = isLatin1() ? StringLatin1.hashCode(value)

: StringUTF16.hashCode(value);

}

**return** h;

}

As you can see in the above code the **String class** has overridden the **hashCode() method**. It returns the int value.

### **Why OR When we should Override the hashCode() method?**

The default implementation of **hashCode() method** is not provided in [**Object class**](https://javagoal.com/object-class-in-java/). It’s a **native method**.  
Before overriding the **hashCode() method**. We should understand what’s the need to override it.

Let’s try to understand with the [**HashSet**](https://javagoal.com/hashset-class/)example. As you already know [**HashSet**](https://javagoal.com/hashset-class/)stores only unique values. Because it uses the **hashCode() method** to store the values in the hash table.  Whenever [**HashSet**](https://javagoal.com/hashset-class/)stores the value, it compares the **hash code**of each object and adds it to the hash table. In Java, there are some classes that override the hashCode() method. We will see how HashSet stores unique **Integer**values by use of **hashCode() method**.

Let’s have a look at the **Integer class**that **overrides** the hashCode() method.

@Override

public int hashCode()

{

**return** Integer.hashCode(value);

}

import java.util.HashSet;

public **class** HashSetExample

{

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

{

HashSet<Integer> setOfInteger = **new** HashSet<Integer>();

setOfInteger.add(1);

setOfInteger.add(2);

setOfInteger.add(1);

System.out.println("Size of HashSet: "+ setOfInteger.size());

}

}

***Output:****Size of HashSet: 2*

In the above example, we have added three Integer objects in [**HashSet**](https://javagoal.com/hashset-class/), but it contains only two objects. How it removes the duplicity of objects?  
**STEP 1:** When we are adding the first object in [**HashSet**](https://javagoal.com/hashset-class/). HashSet stores it inside a bucket and links it to the value of hashcode().

setOfInteger.add(1);

**STEP 2:** Adding another object in [**HashSet**](https://javagoal.com/hashset-class/) and adds value as mentioned in the first step.

setOfInteger.add(2);

**STEP 3:** Now we are adding the same element with the same hash code is inserted into the set. It doesn’t add it. Because [***HashSet***](https://javagoal.com/hashset-class/)makes a search for an element inside it, and it generates the element’s hash code and looks for a bucket that corresponds to this hash code.[**HashSet**](https://javagoal.com/hashset-class/)stores its element in Hash Table by use of the ***unique key.***Itstores each element in memory buckets. Each bucket has a particular hash code.

setOfInteger.add(1);

## ****The Contract Between equals() and hashcode()?****

As per our above discussion, both methods are used for comparison of objects. Both methods should override when we are talking about a huge application that considers two objects as equal when some business fact happens. Even the Object class provides the default implementation, but it is not enough to satisfy business needs.

In this section, we will discuss multiple examples of ***hashCode() and equals() method***. Firstly, we will discuss the problem area.

We will do one example with the default implementation of**hashCode() and equals() method**.

class Book

{

int bookId;

String name;

public int getBookId() {

**return** bookId;

}

public **void** setBookId(int bookId) {

**this**.bookId = bookId;

}

public String getName() {

**return** name;

}

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

**this**.name = name;

}

public Book(int bookId, String name) {

**this**.bookId = bookId;

**this**.name = name;

}

}

public **class** ExampleOfEqualsAndHashCode

{

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

{

Book book1 = **new** Book(123, "Java");

Book book2 = **new** Book(123, "Java");

System.out.println("Is both books are same : "+ book1.equals(book2));

System.out.println("HashCode of book1 :"+ book1.hashCode());

System.out.println("HashCode of book2 :"+ book2.hashCode());

}

}

***Output:****Is both books are same : false  
HashCode of book1 :1556956098  
HashCode of book2 :1252585652*

In the above example, we are defining a class **ExampleOfEqualsAndHashCode**that checks whether two instances of ***Book*** (who have the exact same attributes) are considered as equal.

It is returning **false**because when we are calling **equals() method**then JVM invoking the **equals()  method** that presented in**Object class**with the default implementation.

public boolean equals(Object obj) {

**return** (**this** == obj);

}

As per the observation of default implementation, it is comparing the references of objects. Both instances of **Book**class(**book1 and book2**) have the same attribute values but they are stored in different memory locations. So, they are considered different objects.

**So how to resolve the problem? We can resolve this problem by overriding of equals() method.**

Let’s discuss the same example when we are overriding the **equals() method**. We are considering that two **Books** are equal if they have the same **bookId**, so we override the **equals()** **method**and provide our own implementation.

class Book

{

int bookId;

String name;

public int getBookId() {

**return** bookId;

}

public **void** setBookId(int bookId) {

**this**.bookId = bookId;

}

public String getName() {

**return** name;

}

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

**this**.name = name;

}

public Book(int bookId, String name) {

**this**.bookId = bookId;

**this**.name = name;

}

@Override

public boolean equals(Object obj)

{

**return** **this**.getBookId() == ((Book) obj).getBookId();

}

}

public **class** ExampleOfEqualsAndHashCode

{

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

{

Book book1 = **new** Book(123, "Java");

Book book2 = **new** Book(123, "Java");

System.out.println("Is both books are same : "+ book1.equals(book2));

System.out.println("HashCode of book1 :"+ book1.hashCode());

System.out.println("HashCode of book2 :"+ book2.hashCode());

}

}

***Output:****Is both books are same : true  
HashCode of book1 :1556956098  
HashCode of book2 :1252585652*

In the above implementation, we are overriding the**equals() method**and provide own implementation.  We are considering that two Books are equal if and only if they have the same **bookId**.

It is returning true because when we are calling **equals() method** then **JVM**invoking the **equals()  method** that presented in **Book class** with implementation.

@Override

public boolean equals(Object obj)

{

**return** **this**.getBookId() == ((Book) obj).getBookId();

}

**NOTE: We should override the equals() method with best practices. Because this method creates a lot of impact on performance so we must take care. You can read the best practices for the**[***equals() method***](https://javagoal.com/equals-method-in-java/)**.**

@Override

public boolean equals(Object obj)

{

**if** (obj == **null**) **return** **false**;

**if** (!(obj instanceof Book))

**return** **false**;

**if** (obj == **this**)

**return** **true**;

**return** **this**.getBookId() == ((Book) obj).getBookId();

}

We have seen how we should override the **equals() method** and why we need to override it.

Now we will understand the use of **hashCode()** **method**and why we should override it.

The **hashCode() method**is useful to override when you want to insert the objects into a **HashTable, HashMap, and HashSet.** If you are working inany application that works with hashTable then you must override the **hashCode() method.**If you want to know the working on **HashSet**you can read from [here](https://javagoal.com/how-hashset-working-internally/).

Let’s discuss it with example.

import java.util.HashSet;

**class** Book

{

int bookId;

String name;

public int getBookId() {

**return** bookId;

}

public **void** setBookId(int bookId) {

**this**.bookId = bookId;

}

public String getName() {

**return** name;

}

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

**this**.name = name;

}

public Book(int bookId, String name) {

**this**.bookId = bookId;

**this**.name = name;

}

@Override

public boolean equals(Object obj)

{

**return** **this**.getBookId() == ((Book) obj).getBookId();

}

}

public **class** ExampleOfEqualsAndHashCode

{

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

{

Book book1 = **new** Book(123, "Java");

Book book2 = **new** Book(123, "Java");

HashSet<Book> setOfBook = **new** HashSet<Book>();

setOfBook.add(book1);

setOfBook.add(book2);

System.out.println("Size of HashSet : "+ setOfBook.size());

}

}

***Output:****Size of HashSet : 2*

In the above example, both instances have the same attribute but HashSet considered them different.

#### **Reason of the above problem?**

Because **HashSet** uses the **hashTable**to store the element. When inserting an object into a **hashtable**. Each value is stored on the base of key and the hash code of the key is calculated. The hash code of key is used to determine where to **search** for the object.

**Step1:**So, when we were adding **book1** instance in **HashSet**. Firstly, a unique hashCode was calculated and inserted the object into hashTable. To calculate the hashcode of the key, The JVM invoked the**hashCode() method** of **Object class**.

**Step2:** After adding the **book1**instance, we were adding **book2** instance. It returned different hashCode when JVM tries to calculate the hashCode of the object. Even both objects are the same.

**We can resolve this problem by overriding the hashCode() method.**

import java.util.HashSet;

**class** Book

{

int bookId;

String name;

public int getBookId() {

**return** bookId;

}

public **void** setBookId(int bookId) {

**this**.bookId = bookId;

}

public String getName() {

**return** name;

}

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

**this**.name = name;

}

public Book(int bookId, String name) {

**this**.bookId = bookId;

**this**.name = name;

}

@Override

public boolean equals(Object obj)

{

**return** **this**.getBookId() == ((Book) obj).getBookId();

}

@Override

public int hashCode()

{

**return** bookId;

}}

public **class** ExampleOfEqualsAndHashCode

{

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

{

Book book1 = **new** Book(123, "Java");

Book book2 = **new** Book(123, "Java");

HashSet<Book> setOfBook = **new** HashSet<Book>();

setOfBook.add(book1);

setOfBook.add(book2);

System.out.println("Size of HashSet : "+ setOfBook.size());

}

}

***Output:****Size of HashSet : 1*

# What is the Comparable java interface?

It is an interface that can be used to sort collection elements. This interface is found in **java.lang package** and contains only one method named **compareTo(Object).** It imposes order of all the object of class that implements it.

**NOTE: All the Wrapper classes implement the Comparable interface by default.**

By use of **Comparable interface**, Object can compare itself with another object. You can provide a single sorting sequence only. It means you can compare the Objects of class based on single data members only.  
For example: If you have created a Student class that implements the Comparable interface. So now you can sort the Student data on based on **rollNo**or **name**or **age**etc.

**Why we use comparable?**

Let’s say you have some data of Student and you want to sort them based on their rollNo.  
The first thing comes in mind is that you must compare Student’s rollNo. So, in java you must compare the object with other objects for sorting.

To provide the functionality of Sorting you have to implement the **Comparable interface**. The **Comparable interface** has a method **compareTo()**that is used to compare the objects.

**How to use it?**

First of all, the class must have to implement the **Comparable interface** and provide the body to **compareTo().**

**public int compareTo(Object obj):** This method is used to compare the current object with the specified object. It takes **obj** as a parameter and returns int value.  
It returns:  
**positive integer**, if current object is greater than the specified object.  
**negative integer**, if current object is less than the specified object.  
**zero**, if current object is equal to the specified object.

In java, String and Wrapper class already implements this interface. So that use can directly compare those objects. For User defined class you have implement it and provide the body to **comparerTo() method**.

import java.util.ArrayList;

import java.util.Collections;

**class** Student **implements** Comparable<Student>

{

int rollNo;

String name;

int age;

Student(int rollno,String name,int age)

{

**this**.rollNo=rollno;

**this**.name=name;

**this**.age=age;

}

public int compareTo(Student student)

{

**if**(rollNo == student.rollNo)

**return** 0;

**else** **if**(rollNo > student.rollNo)

**return** 1;

**else**

**return** -1;

}

}

public **class** ExampleOfComparable

{

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

{

ArrayList<Student> listOfStudent = **new** ArrayList<Student>();

listOfStudent.add(**new** Student(1,"Ravi",26));

listOfStudent.add(**new** Student(2,"kant",27));

listOfStudent.add(**new** Student(3,"kamboj",20));

// It Sorts all the objects and internally using the CompareTo method

Collections.sort(listOfStudent);

**for**(Student student:listOfStudent)

{

System.out.println("RollNo of Student = "+student.rollNo);

System.out.println("Age of Student = "+student.age);

System.out.println("Name of Student = "+student.name);

}

}

}

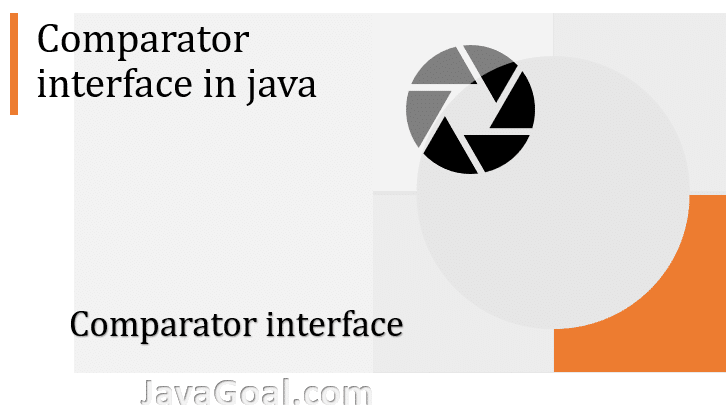
***Output:****RollNo of Student = 1  
Age of Student = 26  
Name of Student = Ravi  
RollNo of Student = 2  
Age of Student = 27  
Name of Student = kant  
RollNo of Student = 3  
Age of Student = 20  
Name of Student = kamboj*

Now, suppose we want sort Student data by their age also. So, you must make changes in **compareTo() method.** But this is not a good approach because requirement can change any time and every time you must make changes in same method. So,  The solution is using [**Comparator** **Interface**](https://javagoal.com/comparator-interface/).

**What is the Comparator** **interface in Java?**

It is an interface that can be used to sort collection elements. This interface is found in **java.util** **package** and contains two methods **compare(Object obj1, Object2 obj)**and**equals(Object obj).** It imposes order of all the object of class that implements it. **Comparator interface** is used to order the objects of a user-defined class.

By use of **Comparator**interface, Object can compare itself with another object. You can provide a multiple sorting sequence. It means you can compare the Objects of class based on multiple data members.  
**For example:** If you have created a **Student class** that implements the **Comparator**interface. So now you can sort the **Student**data on based of **rollNo**or **name**or **age**etc.



**Why we use Comparator?**

In a **Comparable interface,**we must implement the **comparable interface**in the same class for which we want to perform sorting. So, what if we want to change the Sorting data member then we must make changes in the same method of the class. But the **Comparator**is external to the element type we are comparing. We can create multiple separate classes that can implement **Comparator** to compare by different members.

**How to use it?**

Let’s say you have Student data and you want to sort the data based on different data members of the class. For every data member, you can create a separate class that will implement the Comparator interface and provides the body to **compare() method**.

**public int compare(Object obj1, Object obj2):** This method is used to compare the first object(**obj1**) with second object(**obj1**). It takes **two**parameters and returns int value.  
It returns:  
**positive integer**, if first object is greater than the second object.  
**negative integer**, if first current object is less than the second object.  
**zero**, if first object is equal to the second object.

**public boolean equals(Object obj):** You can compare current object with specified object. It returns boolean value based on comparison.

import java.util.ArrayList;

import java.util.Collections;

import java.util.Comparator;

**class** Student

{

int rollNo;

String name;

int age;

Student(int rollno,String name,int age)

{

**this**.rollNo=rollno;

**this**.name=name;

**this**.age=age;

}

}

**class** AgeComparator **implements** Comparator<Student>

{

@Override

public int compare(Student student1, Student student2)

{

**if**(student1.age==student2.age)

**return** 0;

**else** **if**(student1.age>student2.age)

**return** 1;

**else**

**return** -1;

}

}

**class** RollNoComparator **implements** Comparator<Student>

{

@Override

public int compare(Student student1, Student student2)

{

**if**(student1.rollNo==student2.rollNo)

**return** 0;

**else** **if**(student1.rollNo>student2.rollNo)

**return** 1;

**else**

**return** -1;

}

}

public **class** ExampleOfComparable

{

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

{

ArrayList<Student> listOfStudent = **new** ArrayList<Student>();

listOfStudent.add(**new** Student(1,"Ravi",26));

listOfStudent.add(**new** Student(2,"kant",27));

listOfStudent.add(**new** Student(3,"kamboj",20));

// It Sorts all the objects based on Age

Collections.sort(listOfStudent, **new** AgeComparator());

**for**(Student student:listOfStudent)

{

System.out.println("RollNo of Student = "+student.rollNo);

System.out.println("Age of Student = "+student.age);

System.out.println("Name of Student = "+student.name);

}

// It Sorts all the objects based on RollNo

Collections.sort(listOfStudent, **new** RollNoComparator());

**for**(Student student:listOfStudent)

{

System.out.println("RollNo of Student = "+student.rollNo);

System.out.println("Age of Student = "+student.age);

System.out.println("Name of Student = "+student.name);

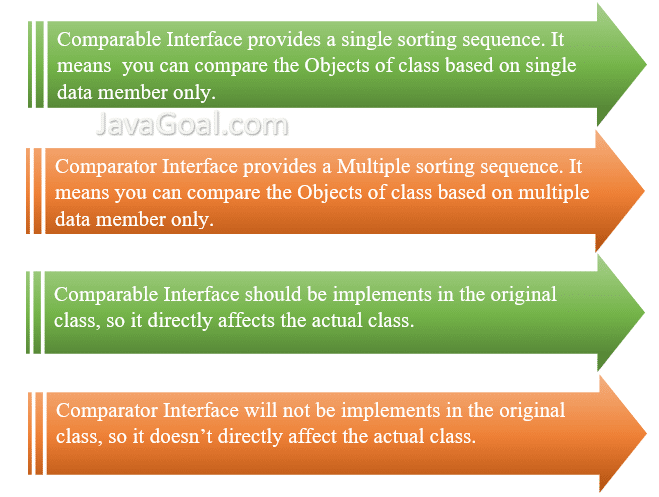
}

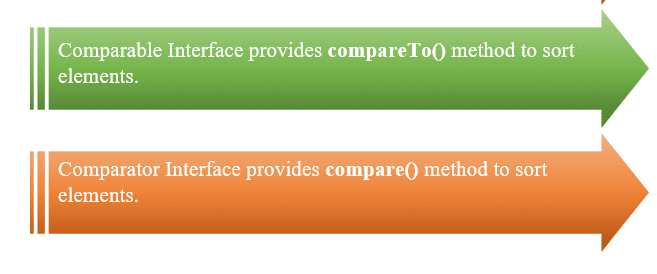
}

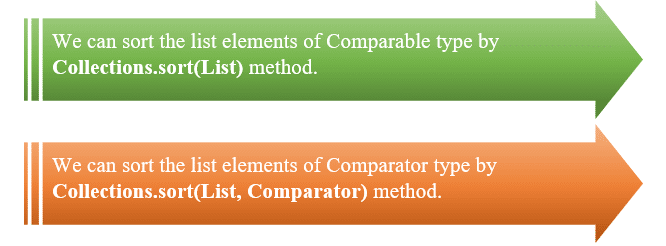
}

***Output:****RollNo of Student = 3  
Age of Student = 20  
Name of Student = kamboj  
RollNo of Student = 1  
Age of Student = 26  
Name of Student = Ravi  
RollNo of Student = 2  
Age of Student = 27  
Name of Student = kant  
RollNo of Student = 1  
Age of Student = 26  
Name of Student = Ravi  
RollNo of Student = 2  
Age of Student = 27  
Name of Student = kant  
RollNo of Student = 3  
Age of Student = 20  
Name of Student = kamboj*

# **Difference between comparable and comparator**







## ****How Comparator useful over comparable****

Let’s see how Comparator useful over comparable, Let’s discuss them with different examples.

## ****comparator vs comparable****

**Changes in Main Class:** As you know Comparable must be implement it in your class for which you want to perform Sorting. So, you need to change your class.

public **class** Student **implements** Comparable

{

// Body of Class

}

For more details of example please [click here](https://javagoal.com/comparable-interface/).  
But to use **Comparator interface**you do not need to change the original class.You can create a separate class for sorting. It provides ease to programmer.For more details of example please [click here](https://javagoal.com/comparator-interface/).

**Single Vs Multiple Sorting:** If you have only single sorting criteria then you should go with Comparable. Here mean of single sort is, you want to Sort your data only base of single data member. Let’s say you want to sort Student data based on roll number only. For more details of example please [click here](https://javagoal.com/comparable-interface/).  
But if you have more than one sorting criteria then you should go with Comparator. Here mean of multiple sort is, you want to Sort your data base of multiple data member. Let’s say you want to sort Student data based on roll number, Age, rank etc. So, you can perform multiple Sorting based on requirements. For more details of example please [click here](https://javagoal.com/comparator-interface/) .

**Flexibility:**As you know Comparable implemented only in main class. So, you can’t add more criteria later. If want to add any criteria later then you have made changes in existing class.  
But by use of you can add sorting criteria later easily because you do not need to modify the existing ones. You have to make another class for new criteria. So, Comparator provide the flexibility.

**User choice:** By use of Comparable you can sort elements to their natural order.  
But by use of Comparator you can sort the order in different from natural order. Here is example how you can change order.

import java.util.ArrayList;

import java.util.Collections;

import java.util.Comparator;

**class** Student

{

int rollNo;

String name;

int age;

Student(int rollno,String name,int age)

{

**this**.rollNo=rollno;

**this**.name=name;

**this**.age=age;

}

}

**class** RollNoComparator **implements** Comparator<Student>

{

@Override

public int compare(Student student1, Student student2)

{

**if**(student1.rollNo==student2.rollNo)

**return** 0;

**else** **if**(student1.rollNo>student2.rollNo)

**return** 1;

**else**

**return** -1;

}

}

public **class** ExampleOfComparable

{

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

{

ArrayList<Student> listOfStudent = **new** ArrayList<Student>();

listOfStudent.add(**new** Student(1,"Ravi",26));

listOfStudent.add(**new** Student(2,"kant",27));

listOfStudent.add(**new** Student(3,"kamboj",20));

// It Sorts all the objects based on RollNo

Collections.sort(listOfStudent, **new** RollNoComparator().reversed());

**for**(Student student:listOfStudent)

{

System.out.println("RollNo of Student = "+student.rollNo);

System.out.println("Age of Student = "+student.age);

System.out.println("Name of Student = "+student.name);

}

}

}

***Output:****RollNo of Student = 3  
Age of Student = 20  
Name of Student = kamboj  
RollNo of Student = 2  
Age of Student = 27  
Name of Student = kant  
RollNo of Student = 1  
Age of Student = 26  
Name of Student = Ravi*

## ****How comparable useful over Comparator?****

What are the different scenarios in which we should use the **comparable and comparator in java**.

**Usage of Java function Arays.sort() and Collection.sort():** If you are using Comparable for Sorting then you can use **Arrays.sort()** and **Collections.sort()** for sorting directly. You do not need to provide a **comparator**. You just need to call the method.

**Provide ease for TreeMap and TreeSet:**Here I want to suggest you please read [TreeMap](https://javagoal.com/treemap-class/) and [TreeSet](https://javagoal.com/treeset-class/). Because TreeMap and TreeSet also use the Comparator and Comparable.  
  
If your implementing Comparable interface in your class, then objects of that class can be used as keys in a TreeMap or as elements in a TreeSet. But if you are using Comparator interface then you have to write separate Comparator and pass it in the constructor of TreeMap or TreeSet. For more details click here ([TreeSet with Comparator](https://javagoal.com/treesetcomparator-comp/) and [TreeMap with Comparator](https://javagoal.com/treemapcomparator-comp/)).  
Now we will see how can use comparable with TreeSet:

import java.util.TreeSet;

**class** SortedString **implements** Comparable<String>

{

String string1;

SortedString(String s)

{

**this**.string1 = s;

}

@Override

public int compareTo(String string2)

{

**return** string1.compareTo(string2);

}

}

public **class** ExampleOfTreeSet

{

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

{

TreeSet<String> listOfNames = **new** TreeSet<String>();

listOfNames.add("RAVI");

listOfNames.add("JAVA");

listOfNames.add("GOAL");

listOfNames.add("SITE");

**for**(String name : listOfNames)

{

System.out.println(name);

}

}

}

***Output:****GOAL  
JAVA  
RAVI  
SITE*

**Number of classes:** As you know to implement the Comparator interface you have provides extra classes. There is some situation where you do not want to write extra class the you should go with Comparable interface. Because Comparable does not require creation of extra classes.  
For more details of example please here for [Comparable](https://javagoal.com/comparable-interface/) and [Comparator](https://javagoal.com/comparator-interface/) .

# **Why Comparable and Comparator are useful?**

In Java, **Comparable and Comparator** interfaces are useful for sorting the collection of objects. Java has some methods the are useful for sorting of primitive types array or Wrapper classes array or list. So, first of all, we will see how we can sort our Primitive data type array and wrapper class array:

import java.util.ArrayList;

import java.util.Arrays;

import java.util.Collections;

import java.util.List;

public **class** SortingOfDifferentObjects

{

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

{

// int array i.e Primitive data types

int[] intArray = {2,9,11,1};

// String array i.e Wrapper class

String stringArray[] = {"BB", "DD", "RR", "LL"};

// Integer array i.e Wrapper class

Integer integerArray[] = {3,8,10,2};

// ArrayList containing wrapper class String

List<String> listOfString = **new** ArrayList<String>();

listOfString.add("A");

listOfString.add("E");

listOfString.add("L");

listOfString.add("B");

// Sorting of int array i.e Primitive data types

Arrays.sort(intArray);

System.out.println(Arrays.toString(intArray));

// Sorting of String array i.e Wrapper class

Arrays.sort(stringArray);

System.out.println(Arrays.toString(stringArray));

// Sorting of Integer array i.e Wrapper class

Arrays.sort(integerArray);

System.out.println(Arrays.toString(integerArray));

// Sorting of ArrayList containing wrapper class String

Collections.sort(listOfString);

System.out.println(listOfString.toString());

}

}

***Output:****[1, 2, 9, 11]  
[BB, DD, LL, RR]  
[2, 3, 8, 10]  
[A, B, E, L]*

In above example you can see how we are sorting the Array of different types. To provide the Sorting functionality for Custom class Java provides two interfaces  
**Java.lang.Comparable** and **java.util.Comparator**

Let’s take an example, suppose you are maintaining the data of College. There are number of Students and you want to collect all the data in ascending order of **rollNo**. So now you need to Sort them. But you can achieve this task by use [comparable](https://javagoal.com/comparable-interface/)and [comparator interface](https://javagoal.com/comparator-interface/). You can see Sorting of custom class in [details](https://javagoal.com/comparable-interface/).

# **How Comparator useful over comparable**

In this post we see **comparator vs comparable** a d when we should you them. How Comparator useful over comparable, Let’s discuss them with different examples.

## ****comparator vs comparable****

**Changes in Main Class:** As you know Comparable must be implement it in your class for which you want to perform Sorting. So, you need to change your class.

public **class** Student **implements** Comparable

{

// Body of Class

}

For more details of example please [click here](https://javagoal.com/comparable-interface/).  
But to use **Comparator interface**you do not need to change the original class.You can create a separate class for sorting. It provides ease to programmer.For more details of example please [click here](https://javagoal.com/comparator-interface/).

**Single Vs Multiple Sorting:** If you have only single sorting criteria then you should go with Comparable. Here mean of single sort is, you want to Sort your data only base of single data member. Let’s say you want to sort Student data based on roll number only. For more details of example please [click here](https://javagoal.com/comparable-interface/).  
But if you have more than one sorting criteria then you should go with Comparator. Here mean of multiple sort is, you want to Sort your data base of multiple data member. Let’s say you want to sort Student data based on roll number, Age, rank etc. So, you can perform multiple Sorting based on requirements. For more details of example please [click here](https://javagoal.com/comparator-interface/) .

**Flexibility:**As you know Comparable implemented only in main class. So, you can’t add more criteria later. If want to add any criteria later then you have made changes in existing class.  
But by use of you can add sorting criteria later easily because you do not need to modify the existing ones. You have to make another class for new criteria. So, Comparator provide the flexibility.

**User choice:** By use of Comparable you can sort elements to their natural order.  
But by use of Comparator you can sort the order in different from natural order. Here is example how you can change order.

import java.util.ArrayList;

import java.util.Collections;

import java.util.Comparator;

**class** Student

{

int rollNo;

String name;

int age;

Student(int rollno,String name,int age)

{

**this**.rollNo=rollno;

**this**.name=name;

**this**.age=age;

}

}

**class** RollNoComparator **implements** Comparator<Student>

{

@Override

public int compare(Student student1, Student student2)

{

**if**(student1.rollNo==student2.rollNo)

**return** 0;

**else** **if**(student1.rollNo>student2.rollNo)

**return** 1;

**else**

**return** -1;

}

}

public **class** ExampleOfComparable

{

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

{

ArrayList<Student> listOfStudent = **new** ArrayList<Student>();

listOfStudent.add(**new** Student(1,"Ravi",26));

listOfStudent.add(**new** Student(2,"kant",27));

listOfStudent.add(**new** Student(3,"kamboj",20));

// It Sorts all the objects based on RollNo

Collections.sort(listOfStudent, **new** RollNoComparator().reversed());

**for**(Student student:listOfStudent)

{

System.out.println("RollNo of Student = "+student.rollNo);

System.out.println("Age of Student = "+student.age);

System.out.println("Name of Student = "+student.name);

}

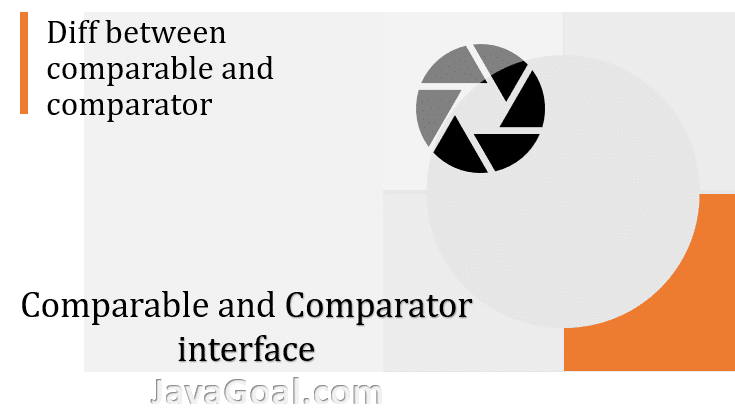
}

}

***Output:****RollNo of Student = 3  
Age of Student = 20  
Name of Student = kamboj  
RollNo of Student = 2  
Age of Student = 27  
Name of Student = kant  
RollNo of Student = 1  
Age of Student = 26  
Name of Student = Ravi*

# **How comparable useful over Comparator?**

In this post, we see the **diff between comparable and comparator** and how comparable useful over Comparator? What are the different scenarios in which we should use the **comparable and comparator in java**.



## ****Diff between comparable and comparator****

**Usage of Java function Arays.sort() and Collection.sort():** If you are using Comparable for Sorting then you can use **Arrays.sort()** and **Collections.sort()** for sorting directly. You do not need to provide a **comparator**. You just need to call the method.

**Provide ease for TreeMap and TreeSet:**Here I want to suggest you please read [TreeMap](https://javagoal.com/treemap-class/) and [TreeSet](https://javagoal.com/treeset-class/). Because TreeMap and TreeSet also use the Comparator and Comparable.  
  
If your implementing Comparable interface in your class, then objects of that class can be used as keys in a TreeMap or as elements in a TreeSet. But if you are using Comparator interface then you have to write separate Comparator and pass it in the constructor of TreeMap or TreeSet. For more details click here ([TreeSet with Comparator](https://javagoal.com/treesetcomparator-comp/) and [TreeMap with Comparator](https://javagoal.com/treemapcomparator-comp/)).

Now we will see how can use comparable with TreeSet:

import java.util.TreeSet;

**class** SortedString **implements** Comparable<String>

{

String string1;

SortedString(String s)

{

**this**.string1 = s;

}

@Override

public int compareTo(String string2)

{

**return** string1.compareTo(string2);

}

}

public **class** ExampleOfTreeSet

{

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

{

TreeSet<String> listOfNames = **new** TreeSet<String>();

listOfNames.add("RAVI");

listOfNames.add("JAVA");

listOfNames.add("GOAL");

listOfNames.add("SITE");

**for**(String name : listOfNames)

{

System.out.println(name);

}

}

}

***Output:****GOAL  
JAVA  
RAVI  
SITE*

# **Number of classes:** As you know to implement the Comparator interface you have provides extra classes. There is some situation where you do not want to write extra class the you should go with Comparable interface. Because Comparable does not require creation of extra classes. **Generics in Java**

Generic is the most important in Java. In this article, we will read all detail of java generics.  
Here is the table content of the article will we will cover this topic.  
**1.**[**What is Java Generics?**](https://javagoal.com/java-generics/#1)  
**2.**[**Generic class**](https://javagoal.com/java-generic-type/) **3.**[**Generic method**](https://javagoal.com/java-generic-type/) **4.**[**Generic Interface**](https://javagoal.com/java-generic-type/) **5.**[**Advantages of Generics**](https://javagoal.com/advantages-of-generics-in-java/)

## ****What is Java Generics?****

In Java, we can create a Generic class, Generic method, and Generic Interface that accepts the input to make it as a specific type. They can accept any type of data according to requirements.

Suppose you want to perform a task with three different [data types](https://javagoal.com/data-types-in-java/) of parameters.  
You want to show data of String, int, and float type. You must create three functions that accept different inputs and perform the same task.

First of all, we will discuss it without a generic concept. Here we need to create three functions.

public **class** ExampleWithoutGeneric

{

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

{

ExampleWithoutGeneric obj = **new** ExampleWithoutGeneric();

// Showing data of integer

System.out.println("Calling by int values");

obj.showIntData(5, 10);

// Showing data of String

System.out.println("Calling by String values");

obj.showStringData("Hello", "Java");

// Showing data of float

System.out.println("Calling by float values");

obj.showFloatData(1.23f, 3.1f);

}

public **void** showIntData(int a, int b)

{

System.out.println(a);

System.out.println(b);

}

public **void** showStringData(String a, String b)

{

System.out.println(a);

System.out.println(b);

}

public **void** showFloatData(float a, float b)

{

System.out.println(a);

System.out.println(b);

}

}

***Output:****Calling by int values  
5  
10  
Calling by String values  
Hello  
Java  
Calling by float values  
1.23  
3.1*

Now we will see, how we can do it with Generics.

public **class** ExampleWithGeneric

{

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

{

ExampleWithGeneric obj = **new** ExampleWithGeneric();

System.out.println("Calling by integer values");

obj.showData(5, 10);

System.out.println("Calling by String values");

obj.showData("Hello", "Java");

System.out.println("Calling by float values");

obj.showData(5.1f, 10.2f);

}

public <T> **void** showData(T a, T b)

{

System.out.println(a);

System.out.println(b);

}

}

***Output:****Calling by integer values  
5  
10  
Calling by String values  
Hello  
Java  
Calling by float values  
5.1  
10.2*

In this example, we can perform the task by the use of only a single method. It is a generic method that can accept any type of input. As of now do not focus on syntax we will discuss it in detail.

As you all familiar with the Collection framework. The Collection framework provides generic classes. Let’s take an example of the ArrayList class. ArrayList is a generic class because it can contain and return any type of data. Let’s see the syntax of ArrayList:

ArrayList<String> listOfString = **new** ArrayList<String>();

ArrayList<Integer> listOfInteger = **new** ArrayList<Integer>();

As you can see the ArrayList class can take any type of data.

**Java generic type parameter**

Here is the naming convention provided by Java that helps you to understand the Java Generic. The most used type parameter names are:  
E – Element (Used in Collections Framework for example Set)  
K – Key (Used in Collections Framework  for example Map)  
N – Number  
T – Type  
V – Value (Used in Collections Framework  for Map)

# **Generic types in Java**

Now we have some knowledge of [Generics in Java](https://javagoal.com/java-generics/). So, we should start exploring other important concepts revolving around generics. We will discuss the Java generic type.

**1.**[**Generic class**](https://javagoal.com/java-generic-type/#1)  
**2.**[**How to create a generic class?**](https://javagoal.com/java-generic-type/#2) **3.**[**Generic method**](https://javagoal.com/java-generic-type/#3) **4.**[**Generic Interface**](https://javagoal.com/java-generic-type/#4)

## ****1. Generic class****

Without generic, A class that can refer to any type of data. When you create a class, it can accept any type of data. Let’s understand with an example:

class Example

{

private Object obj;

public **void** set(Object t) { **this**.obj = t; }

public Object get() { **return** obj; }

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

{

Example exampleOne = **new** Example();

exampleOne.set(123);

Integer intValue = (Integer)exampleOne.get();

System.out.println(intValue);

Example exampleTwo = **new** Example();

exampleTwo.set("Hello");

Integer intValueTwo = (Integer)exampleTwo.get();

System.out.println(intValueTwo);

}

}

***Output:****123  
Exception in thread “main” java.lang.ClassCastException: class java.lang.String cannot be cast to class java.lang.Integer (java.lang.String and java.lang.Integer are in module java.base of loader ‘bootstrap’)  
at Example.main(Example.java:18)*

It throwing a **ClassCastException**because we are casting String to Integer. To resolve the problem of runtime exception we can use generic class. That will accept only one type of data.

Let’s say you want to initialize the class with a certain type. It means the class should be used for only one type. So, you can make it a **generic class**. The **generic class** can take the object of a specific type. It is part of Java generic type.

## ****How to create a generic class?****

We use <T> to specify parameter types in generic class creation. By use of <T>, you can make a class of specific types.

class Example<T>

{

private T obj;

public **void** set(T t) { **this**.obj = t; }

public T get() { **return** obj; }

}

class Example<T>

{

private T obj;

public **void** set(T t) { **this**.obj = t; }

public T get() { **return** obj; }

}

public **class** Student

{

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

{

Example<Integer> example = **new** Example<Integer>();

example.set(1);

example.add("JavaGoal"); //Compile time error

System.out.println(example.get());

}

}

***Output:****Exception in thread “main” java.lang.Error: Unresolved compilation problem: The method add(String) is undefined for the type Example<Integer> at Student.main(Student.java:16)*

In the above example, we want the object of the class to hold the value of String type only. So, when we trying to add an object of Integer it is showing compilation error. It applies the restriction to the programmer, So that programmer can add only one type of data and ignore the run time exception. By the use of generic,the compiler shows the errors at compile time rather than the run time. It saves the programmer time because it’s difficult to find the error on runtime. Its always better to find the errors at compile time rather than the run time.

class Test<T1, T2>

{

T1 obj1; // An object of type T1

T2 obj2; // An object of type U2

public **void** setData(T1 object1, T2 object2)

{

obj1 = object1;

obj2 = object2;

}

// To print objects of T1 and T2

public **void** print()

{

System.out.println(obj1);

System.out.println(obj2);

}

}

**class** MainClass

{

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

{

Test<String, Integer> test = **new** Test<String, Integer>();

test.setData("Hello", 123);

test.print();

}

}

***Output:****Hello  
123*

Onward java 5, Collection framework supports the generic concept. You can see the example of ArrayList.

import java.util.ArrayList;

**class** ExampleOfGeneric

{

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

{

ArrayList<String> listOfNames = **new** ArrayList<String>();

listOfNames.add("JAVA");

listOfNames.add("GOAL");

//list.add(123);//compile time error

**for** (String s : listOfNames)

{

System.out.println(s);

}

}

}

***Output:****JAVA  
GOAL*

## ****2. Generic method****

We can create a generic method that can accept any type of argument. These are similar to the generic class but In the Generic method, the scope of the parameter is within the method.

public **class** Example

{

public <T> **void** printData(T data)

{

System.out.println("Data is = "+ data);

}

public <T> T checkData(T data)

{

**return** data;

}

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

{

Example obj = **new** Example();

// Using int as parameter

obj.checkData(1);

obj.printData(1);

// Using String as parameter

obj.checkData("JAVAGOAL");

obj.printData("JAVAGOAL");

}

}

***Output:****Data is = 1  
Data is = JAVAGOAL*

In the above example, we created to the method of the generic type. Both can take any type of data. We are proving int and String data as a parameter to generic methods.

## ****3. Generic Interface****

Like a generic class, Let’s say you want to initialize the Interface with a certain type. It means the Interface should be used for only one type. So, you can make it a generic Interface. The generic interface can take an object of a specific type. You can create a generic interface by use <T> to specify parameter types in generic Interface creation.

//Generic interface definition

interface ExampleInterface<T>

{

public **void** ShowDataCal(T t);

}

//A class implementing generic interface

**class** DemoClass **implements** ExampleInterface<String>

{

public **void** ShowDataCal(String t)

{

//some code

}

}

You can take the example of a [Comparable interface](https://javagoal.com/data-types-in-java/). It is the best example of Generics in interfaces.

public interface Comparable<T> {

public int compareTo(T o);

}

# PASTE HERE