Lambda Expression:

The Lambda expression is used to provide the implementation of an interface which has functional interface. It saves a lot of code. In case of lambda expression, we don't need to define the method again for providing the implementation. Here, we just write the implementation code.

Java lambda expression is treated as a function, so compiler does not create .class file.

But, before getting into lambdas, we first need to understand functional interfaces.

**What is Functional Interface?**

If a Java interface contains one and only one abstract method then it is termed as functional interface.

For example, the Runnable interface from package java.lang; is a functional interface because it constitutes only one method i.e. run().

### Example 1: Define a Functional Interface in java

import java.lang.FunctionalInterface;

@FunctionalInterface

public interface MyInterface{

// the single abstract method

double getValue();

}

In the above example, the interface MyInterface has only one abstract method getValue(). Hence, it is a functional interface.

Here, we have used the annotation @FunctionalInterface. The annotation forces the Java compiler to indicate that the interface is a functional interface. Hence, does not allow to have more than one abstract method.

## Why use Lambda Expression

1. To provide the implementation of Functional interface.
2. Less coding.

Java Lambda Expression Syntax

(argument-list) -> {body}

Java lambda expression is consisted of three components.

**1) Argument-list:** It can be empty or non-empty as well.

**2) Arrow-token:** It is used to link arguments-list and body of expression.

**3) Body:** It contains expressions and statements for lambda expression.

**No Parameter Syntax**

1. () -> {
2. //Body of no parameter lambda
3. }

**One Parameter Syntax**

1. (p1) -> {
2. //Body of single parameter lambda
3. }

**Two Parameter Syntax**

1. (p1,p2) -> {
2. //Body of multiple parameter lambda
3. }

Let's see a scenario where we are not implementing Java lambda expression. Here, we are implementing an interface without using lambda expression.

Without Lambda Expression

1. **interface** Drawable{
2. **public** **void** draw();
3. }
4. **public** **class** LambdaExpressionExample {
5. **public** **static** **void** main(String[] args) {
6. **int** width=10;
8. //without lambda, Drawable implementation using anonymous class
9. Drawable1 d=**new** Drawable1(){
10. **public** **void** draw(){
11. System.out.println("Drawing "+width);
12. }
13. };
14. d.draw();
15. }
16. }

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

Output:

Drawing 10

Java Lambda Expression Example

Now, we are going to implement the above example with the help of Java lambda expression.

1. @FunctionalInterface  //It is optional
2. **interface** Drawable{
3. **public** **void** draw();
4. }
6. **public** **class** LambdaExpressionExample2 {
7. **public** **static** **void** main(String[] args) {
8. **int** width=10;
10. //with lambda
11. Drawable d2=()->{
12. System.out.println("Drawing "+width);
13. };
14. d2.draw();
15. }
16. }

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=LambdaExpressionExample2)

Output:

Drawing 10

A lambda expression can have zero or any number of arguments. Let's see the examples:

Java Lambda Expression Example: No Parameter

1. **interface** Sayable{
2. **public** String say();
3. }
4. **public** **class** LambdaExpressionExample3{
5. **public** **static** **void** main(String[] args) {
6. Sayable s=()->{
7. **return** "I have nothing to say.";
8. };
9. System.out.println(s.say());
10. }
11. }

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=LambdaExpressionExample3)

Output:

I have nothing to say.

Java Lambda Expression Example: Single Parameter

1. **interface** Sayable{
2. **public** String say(String name);
3. }
4. **public** **class** LambdaExpressionExample4{
5. **public** **static** **void** main(String[] args) {
7. // Lambda expression with single parameter.
8. Sayable s1=(name)->{
9. **return** "Hello, "+name;
10. };
11. System.out.println(s1.say("Sonoo"));
13. // You can omit function parentheses
14. Sayable s2= name ->{
15. **return** "Hello, "+name;
16. };
17. System.out.println(s2.say("Sonoo"));
18. }
19. }

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=LambdaExpressionExample4)

Output:

Hello, Sonoo

Hello, Sonoo

Java Lambda Expression Example: Multiple Parameters

1. **interface** Addable{
2. **int** add(**int** a,**int** b);
3. }
5. **public** **class** LambdaExpressionExample5{
6. **public** **static** **void** main(String[] args) {
8. // Multiple parameters in lambda expression
9. Addable ad1=(a,b)->(a+b);
10. System.out.println(ad1.add(10,20));
12. // Multiple parameters with data type in lambda expression
13. Addable ad2=(**int** a,**int** b)->(a+b);
14. System.out.println(ad2.add(100,200));
15. }
16. }

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=LambdaExpressionExample5)

Output:

30

300

Java Lambda Expression Example: with or without return keyword

In Java lambda expression, if there is only one statement, you may or may not use return keyword. You must use return keyword when lambda expression contains multiple statements.

1. **interface** Addable{
2. **int** add(**int** a,**int** b);
3. }
5. **public** **class** LambdaExpressionExample6 {
6. **public** **static** **void** main(String[] args) {
8. // Lambda expression without return keyword.
9. Addable ad1=(a,b)->(a+b);
10. System.out.println(ad1.add(10,20));
12. // Lambda expression with return keyword.
13. Addable ad2=(**int** a,**int** b)->{
14. **return** (a+b);
15. };
16. System.out.println(ad2.add(100,200));
17. }
18. }

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=LambdaExpressionExample6)

Output:

30

300

Java Lambda Expression Example: Foreach Loop

1. **import** java.util.\*;
2. **public** **class** LambdaExpressionExample7{
3. **public** **static** **void** main(String[] args) {
5. List<String> list=**new** ArrayList<String>();
6. list.add("ankit");
7. list.add("mayank");
8. list.add("irfan");
9. list.add("jai");
11. list.forEach(
12. (n)->System.out.println(n)
13. );
14. }
15. }

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=LambdaExpressionExample7)

Output:

ankit

mayank

irfan

jai

Java Lambda Expression Example: Multiple Statements

1. @FunctionalInterface
2. **interface** Sayable{
3. String say(String message);
4. }
6. **public** **class** LambdaExpressionExample8{
7. **public** **static** **void** main(String[] args) {
9. // You can pass multiple statements in lambda expression
10. Sayable person = (message)-> {
11. String str1 = "I would like to say, ";
12. String str2 = str1 + message;
13. **return** str2;
14. };
15. System.out.println(person.say("time is precious."));
16. }
17. }

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=LambdaExpressionExample8)

Output:

I would like to say, time is precious.

Java Lambda Expression Example: Creating Thread

You can use lambda expression to run thread. In the following example, we are implementing run method by using lambda expression.

1. **public** **class** LambdaExpressionExample9{
2. **public** **static** **void** main(String[] args) {
4. //Thread Example without lambda
5. Runnable r1=**new** Runnable(){
6. **public** **void** run(){
7. System.out.println("Thread1 is running...");
8. }
9. };
10. Thread t1=**new** Thread(r1);
11. t1.start();
12. //Thread Example with lambda
13. Runnable r2=()->{
14. System.out.println("Thread2 is running...");
15. };
16. Thread t2=**new** Thread(r2);
17. t2.start();
18. }
19. }

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=LambdaExpressionExample9)

Output:

Thread1 is running...

Thread2 is running...

Java lambda expression can be used in the collection framework. It provides efficient and concise way to iterate, filter and fetch data. Following are some lambda and collection examples provided.

Java Lambda Expression Example: Comparator

1. **import** java.util.ArrayList;
2. **import** java.util.Collections;
3. **import** java.util.List;
4. **class** Product{
5. **int** id;
6. String name;
7. **float** price;
8. **public** Product(**int** id, String name, **float** price) {
9. **super**();
10. **this**.id = id;
11. **this**.name = name;
12. **this**.price = price;
13. }
14. }
15. **public** **class** LambdaExpressionExample10{
16. **public** **static** **void** main(String[] args) {
17. List<Product> list=**new** ArrayList<Product>();
19. //Adding Products
20. list.add(**new** Product(1,"HP Laptop",25000f));
21. list.add(**new** Product(3,"Keyboard",300f));
22. list.add(**new** Product(2,"Dell Mouse",150f));
24. System.out.println("Sorting on the basis of name...");
25. // implementing lambda expression
26. Collections.sort(list,(p1,p2)->{
27. **return** p1.name.compareTo(p2.name);
28. });
29. **for**(Product p:list){
30. System.out.println(p.id+" "+p.name+" "+p.price);
31. }
33. }
34. }

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=LambdaExpressionExample10)

Output:

Sorting on the basis of name...

2 Dell Mouse 150.0

1 HP Laptop 25000.0

3 Keyboard 300.0

Java Lambda Expression Example: Filter Collection Data

1. **import** java.util.ArrayList;
2. **import** java.util.List;
3. **import** java.util.stream.Stream;
4. **class** Product{
5. **int** id;
6. String name;
7. **float** price;
8. **public** Product(**int** id, String name, **float** price) {
9. **super**();
10. **this**.id = id;
11. **this**.name = name;
12. **this**.price = price;
13. }
14. }
15. **public** **class** LambdaExpressionExample11{
16. **public** **static** **void** main(String[] args) {
17. List<Product> list=**new** ArrayList<Product>();
18. list.add(**new** Product(1,"Samsung A5",17000f));
19. list.add(**new** Product(3,"Iphone 6S",65000f));
20. list.add(**new** Product(2,"Sony Xperia",25000f));
21. list.add(**new** Product(4,"Nokia Lumia",15000f));
22. list.add(**new** Product(5,"Redmi4 ",26000f));
23. list.add(**new** Product(6,"Lenevo Vibe",19000f));
25. // using lambda to filter data
26. Stream<Product> filtered\_data = list.stream().filter(p -> p.price > 20000);
28. // using lambda to iterate through collection
29. filtered\_data.forEach(
30. product -> System.out.println(product.name+": "+product.price)
31. );
32. }
33. }

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=LambdaExpressionExample11)

Output:

Iphone 6S: 65000.0

Sony Xperia: 25000.0

Redmi4 : 26000.0

# **Generics in Java**

The **Java Generics** programming is introduced in J2SE 5 to deal with type-safe objects. It makes the code stable by detecting the bugs at compile time.

Before generics, we can store any type of objects in the collection, i.e., non-generic. Now generics force the java programmer to store a specific type of objects.

## Advantage of Java Generics

There are mainly 3 advantages of generics. They are as follows:

**1) Type-safety:** We can hold only a single type of objects in generics. It doesn?t allow to store other objects.

Nested Structure in CKeep Watching

Without Generics, we can store any type of objects.

1. List list = **new** ArrayList();
2. list.add(10);
3. list.add("10");
4. With Generics, it is required to specify the type of object we need to store.
5. List<Integer> list = **new** ArrayList<Integer>();
6. list.add(10);
7. list.add("10");// compile-time error

**2) Type casting is not required:** There is no need to typecast the object.

Before Generics, we need to type cast.

1. List list = **new** ArrayList();
2. list.add("hello");
3. String s = (String) list.get(0);//typecasting
4. After Generics, we don't need to typecast the object.
5. List<String> list = **new** ArrayList<String>();
6. list.add("hello");
7. String s = list.get(0);

**3) Compile-Time Checking:** It is checked at compile time so problem will not occur at runtime. The good programming strategy says it is far better to handle the problem at compile time than runtime.

1. List<String> list = **new** ArrayList<String>();
2. list.add("hello");
3. list.add(32);//Compile Time Error

**Syntax** to use generic collection

1. ClassOrInterface<Type>

**Example** to use Generics in java

1. ArrayList<String>

## Full Example of Generics in Java

Here, we are using the ArrayList class, but you can use any collection class such as ArrayList, LinkedList, HashSet, TreeSet, HashMap, Comparator etc.

1. **import** java.util.\*;
2. **class** TestGenerics1{
3. **public** **static** **void** main(String args[]){
4. ArrayList<String> list=**new** ArrayList<String>();
5. list.add("rahul");
6. list.add("jai");
7. //list.add(32);//compile time error
9. String s=list.get(1);//type casting is not required
10. System.out.println("element is: "+s);
12. Iterator<String> itr=list.iterator();
13. **while**(itr.hasNext()){
14. System.out.println(itr.next());
15. }
16. }
17. }
18. **import** java.util.\*;
19. **class** TestGenerics1{
20. **public** **static** **void** main(String args[]){
21. ArrayList<String> list=**new** ArrayList<String>();
22. list.add("rahul");
23. list.add("jai");
24. //list.add(32);//compile time error
26. String s=list.get(1);//type casting is not required
27. System.out.println("element is: "+s);
29. Iterator<String> itr=list.iterator();
30. **while**(itr.hasNext()){
31. System.out.println(itr.next());
32. }
33. }
34. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestGenerics1)

**Output:**

element is: jai

rahul

jai

## Example of Java Generics using Map

Now we are going to use map elements using generics. Here, we need to pass key and value. Let us understand it by a simple example:

1. **import** java.util.\*;
2. **class** TestGenerics2{
3. **public** **static** **void** main(String args[]){
4. Map<Integer,String> map=**new** HashMap<Integer,String>();
5. map.put(1,"vijay");
6. map.put(4,"umesh");
7. map.put(2,"ankit");
9. //Now use Map.Entry for Set and Iterator
10. Set<Map.Entry<Integer,String>> set=map.entrySet();
12. Iterator<Map.Entry<Integer,String>> itr=set.iterator();
13. **while**(itr.hasNext()){
14. Map.Entry e=itr.next();//no need to typecast
15. System.out.println(e.getKey()+" "+e.getValue());
16. }
18. }}

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestGenerics2)

**Output**

1 vijay

2 ankit

4 umesh

## Generic class

A class that can refer to any type is known as a generic class. Here, we are using the T type parameter to create the generic class of specific type.

Let's see a simple example to create and use the generic class.

### **Creating a generic class:**

1. ***class****MyGen<T>{*
2. *T obj;*

***v*oid** add(T obj){**this**.obj=obj;}

1. T get(){**return** obj;}
2. }

The T type indicates that it can refer to any type (like String, Integer, and Employee). The type you specify for the class will be used to store and retrieve the data.

### **Using generic class:**

Let's see the code to use the generic class.

1. **class** TestGenerics3{
2. **public** **static** **void** main(String args[]){
3. MyGen<Integer> m=**new** MyGen<Integer>();
4. m.add(2);
5. //m.add("vivek");//Compile time error
6. System.out.println(m.get());
7. }}

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestGenerics3)

**Output**

2

## Type Parameters

The type parameters naming conventions are important to learn generics thoroughly. The common type parameters are as follows:

1. T - Type
2. E - Element
3. K - Key
4. N - Number
5. V - Value

## Generic Method

Like the generic class, we can create a generic method that can accept any type of arguments. Here, the scope of arguments is limited to the method where it is declared. It allows static as well as non-static methods.

Let's see a simple example of java generic method to print array elements. We are using here **E** to denote the element.

1. **public** **class** TestGenerics4{
3. **public** **static** < E > **void** printArray(E[] elements) {
4. **for** ( E element : elements){
5. System.out.println(element );
6. }
7. System.out.println();
8. }
9. **public** **static** **void** main( String args[] ) {
10. Integer[] intArray = { 10, 20, 30, 40, 50 };
11. Character[] charArray = { 'J', 'A', 'V', 'A', 'T','P','O','I','N','T' };
13. System.out.println( "Printing Integer Array" );
14. printArray( intArray  );
16. System.out.println( "Printing Character Array" );
17. printArray( charArray );
18. }
19. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestGenerics4)

**Output**

Printing Integer Array

10

20

30

40

50

Printing Character Array

J

A

V

A

T

P

O

I

N

T

## Wildcard in Java Generics

The ? (question mark) symbol represents the wildcard element. It means any type. If we write <? extends Number>, it means any child class of Number, e.g., Integer, Float, and double. Now we can call the method of Number class through any child class object.

We can use a wildcard as a **type of a parameter, field, return type, or local variable. However, it is not allowed to use a wildcard as a type argument for a generic method invocation, a generic class instance creation, or a supertype**.

Let's understand it by the example given below:

1. **import** java.util.\*;
2. **abstract** **class** Shape{
3. **abstract** **void** draw();
4. }
5. **class** Rectangle **extends** Shape{
6. **void** draw(){System.out.println("drawing rectangle");}
7. }
8. **class** Circle **extends** Shape{
9. **void** draw(){System.out.println("drawing circle");}
10. }
11. **class** GenericTest{
12. //creating a method that accepts only child class of Shape
13. **public** **static** **void** drawShapes(List<? **extends** Shape> lists){
14. **for**(Shape s:lists){
15. s.draw();//calling method of Shape class by child class instance
16. }
17. }
18. **public** **static** **void** main(String args[]){
19. List<Rectangle> list1=**new** ArrayList<Rectangle>();
20. list1.add(**new** Rectangle());
22. List<Circle> list2=**new** ArrayList<Circle>();
23. list2.add(**new** Circle());
24. list2.add(**new** Circle());
26. drawShapes(list1);
27. drawShapes(list2);
28. }}

**Output**

drawing rectangle

drawing circle

drawing circle

### **Upper Bounded Wildcards**

The purpose of upper bounded wildcards is to decrease the restrictions on a variable. It restricts the unknown type to be a specific type or a subtype of that type. It is used by declaring wildcard character ("?") followed by the extends (in case of, class) or implements (in case of, interface) keyword, followed by its upper bound.

### **Syntax**

1. List<? **extends** Number>

Here,

**?** is a wildcard character.

**extends**, is a keyword.

**Number**, is a class present in java.lang package

Suppose, we want to write the method for the list of Number and its subtypes (like Integer, Double). Using **List<? extends Number>** is suitable for a list of type Number or any of its subclasses whereas **List<Number>** works with the list of type Number only. So, **List<? extends Number>** is less restrictive than **List<Number>**.

### **Example of Upper Bound Wildcard**

In this example, we are using the upper bound wildcards to write the method for List<Integer> and List<Double>.

1. **import** java.util.ArrayList;
3. **public** **class** UpperBoundWildcard {

6. **private** **static** Double add(ArrayList<? **extends** Number> num) {
8. **double** sum=0.0;
10. **for**(Number n:num)
11. {
12. sum = sum+n.doubleValue();
13. }
15. **return** sum;
16. }
18. **public** **static** **void** main(String[] args) {
20. ArrayList<Integer> l1=**new** ArrayList<Integer>();
21. l1.add(10);
22. l1.add(20);
23. System.out.println("displaying the sum= "+add(l1));
25. ArrayList<Double> l2=**new** ArrayList<Double>();
26. l2.add(30.0);
27. l2.add(40.0);
28. System.out.println("displaying the sum= "+add(l2));

31. }
33. }

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

**Output**

displaying the sum= 30.0

displaying the sum= 70.0

### **Unbounded Wildcards**

The unbounded wildcard type represents the list of an unknown type such as List<?>. This approach can be useful in the following scenarios: -

* When the given method is implemented by using the functionality provided in the Object class.
* When the generic class contains the methods that don't depend on the type parameter.

### **Example of Unbounded Wildcards**

1. **import** java.util.Arrays;
2. **import** java.util.List;
4. **public** **class** UnboundedWildcard {
6. **public** **static** **void** display(List<?> list)
7. {
9. **for**(Object o:list)
10. {
11. System.out.println(o);
12. }
14. }

17. **public** **static** **void** main(String[] args) {
19. List<Integer> l1=Arrays.asList(1,2,3);
20. System.out.println("displaying the Integer values");
21. display(l1);
22. List<String> l2=Arrays.asList("One","Two","Three");
23. System.out.println("displaying the String values");
24. display(l2);
25. }
27. }

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

**Output**

displaying the Integer values

1

2

3

displaying the String values

One

Two

Three

## Lower Bounded Wildcards

The purpose of lower bounded wildcards is to restrict the unknown type to be a specific type or a supertype of that type. It is used by declaring wildcard character ("?") followed by the super keyword, followed by its lower bound.

### **Syntax**

1. List<? **super** Integer>

Here,

**?** is a wildcard character.

**super**, is a keyword.

**Integer**, is a wrapper class.

Suppose, we want to write the method for the list of Integer and its supertype (like Number, Object). Using **List<? super Integer>** is suitable for a list of type Integer or any of its superclasses whereas **List<Integer>** works with the list of type Integer only. So, **List<? super Integer>** is less restrictive than **List<Integer>**.

### **Example of Lower Bound Wildcard**

In this example, we are using the lower bound wildcards to write the method for List<Integer> and List<Number>.

1. **import** java.util.Arrays;
2. **import** java.util.List;
4. **public** **class** LowerBoundWildcard {
6. **public** **static** **void** addNumbers(List<? **super** Integer> list) {
8. **for**(Object n:list)
9. {
10. System.out.println(n);
11. }


15. }
16. **public** **static** **void** main(String[] args) {
18. List<Integer> l1=Arrays.asList(1,2,3);
19. System.out.println("displaying the Integer values");
20. addNumbers(l1);
22. List<Number> l2=Arrays.asList(1.0,2.0,3.0);
23. System.out.println("displaying the Number values");
24. addNumbers(l2);
25. }
27. }

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

**Output**

displaying the Integer values

1

2

3

displaying the Number values

1.0

2.0

3.0

Stream API:

# **Generics in Java**

The **Java Generics** programming is introduced in J2SE 5 to deal with type-safe objects. It makes the code stable by detecting the bugs at compile time.

Before generics, we can store any type of objects in the collection, i.e., non-generic. Now generics force the java programmer to store a specific type of objects.

## Advantage of Java Generics

There are mainly 3 advantages of generics. They are as follows:

**1) Type-safety:** We can hold only a single type of objects in generics. It doesn?t allow to store other objects.

Nested Structure in CKeep Watching

Without Generics, we can store any type of objects.

1. List list = **new** ArrayList();
2. list.add(10);
3. list.add("10");
4. With Generics, it is required to specify the type of object we need to store.
5. List<Integer> list = **new** ArrayList<Integer>();
6. list.add(10);
7. list.add("10");// compile-time error

**2) Type casting is not required:** There is no need to typecast the object.

Before Generics, we need to type cast.

1. List list = **new** ArrayList();
2. list.add("hello");
3. String s = (String) list.get(0);//typecasting
4. After Generics, we don't need to typecast the object.
5. List<String> list = **new** ArrayList<String>();
6. list.add("hello");
7. String s = list.get(0);

**3) Compile-Time Checking:** It is checked at compile time so problem will not occur at runtime. The good programming strategy says it is far better to handle the problem at compile time than runtime.

1. List<String> list = **new** ArrayList<String>();
2. list.add("hello");
3. list.add(32);//Compile Time Error

**Syntax** to use generic collection

1. ClassOrInterface<Type>

**Example** to use Generics in java

1. ArrayList<String>

## Full Example of Generics in Java

Here, we are using the ArrayList class, but you can use any collection class such as ArrayList, LinkedList, HashSet, TreeSet, HashMap, Comparator etc.

1. **import** java.util.\*;
2. **class** TestGenerics1{
3. **public** **static** **void** main(String args[]){
4. ArrayList<String> list=**new** ArrayList<String>();
5. list.add("rahul");
6. list.add("jai");
7. //list.add(32);//compile time error
9. String s=list.get(1);//type casting is not required
10. System.out.println("element is: "+s);
12. Iterator<String> itr=list.iterator();
13. **while**(itr.hasNext()){
14. System.out.println(itr.next());
15. }
16. }
17. }
18. **import** java.util.\*;
19. **class** TestGenerics1{
20. **public** **static** **void** main(String args[]){
21. ArrayList<String> list=**new** ArrayList<String>();
22. list.add("rahul");
23. list.add("jai");
24. //list.add(32);//compile time error
26. String s=list.get(1);//type casting is not required
27. System.out.println("element is: "+s);
29. Iterator<String> itr=list.iterator();
30. **while**(itr.hasNext()){
31. System.out.println(itr.next());
32. }
33. }
34. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestGenerics1)

**Output:**

element is: jai

rahul

jai

## Example of Java Generics using Map

Now we are going to use map elements using generics. Here, we need to pass key and value. Let us understand it by a simple example:

1. **import** java.util.\*;
2. **class** TestGenerics2{
3. **public** **static** **void** main(String args[]){
4. Map<Integer,String> map=**new** HashMap<Integer,String>();
5. map.put(1,"vijay");
6. map.put(4,"umesh");
7. map.put(2,"ankit");
9. //Now use Map.Entry for Set and Iterator
10. Set<Map.Entry<Integer,String>> set=map.entrySet();
12. Iterator<Map.Entry<Integer,String>> itr=set.iterator();
13. **while**(itr.hasNext()){
14. Map.Entry e=itr.next();//no need to typecast
15. System.out.println(e.getKey()+" "+e.getValue());
16. }
18. }}

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestGenerics2)

**Output**

1 vijay

2 ankit

4 umesh

## Generic class

A class that can refer to any type is known as a generic class. Here, we are using the T type parameter to create the generic class of specific type.

Let's see a simple example to create and use the generic class.

### **Creating a generic class:**

1. **class** MyGen<T>{
2. T obj;
3. **void** add(T obj){**this**.obj=obj;}
4. T get(){**return** obj;}
5. }

The T type indicates that it can refer to any type (like String, Integer, and Employee). The type you specify for the class will be used to store and retrieve the data.

### **Using generic class:**

Let's see the code to use the generic class.

1. **class** TestGenerics3{
2. **public** **static** **void** main(String args[]){
3. MyGen<Integer> m=**new** MyGen<Integer>();
4. m.add(2);
5. //m.add("vivek");//Compile time error
6. System.out.println(m.get());
7. }}

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestGenerics3)

**Output**

2

## Type Parameters

The type parameters naming conventions are important to learn generics thoroughly. The common type parameters are as follows:

1. T - Type
2. E - Element
3. K - Key
4. N - Number
5. V - Value

## Generic Method

Like the generic class, we can create a generic method that can accept any type of arguments. Here, the scope of arguments is limited to the method where it is declared. It allows static as well as non-static methods.

Let's see a simple example of java generic method to print array elements. We are using here **E** to denote the element.

1. **public** **class** TestGenerics4{
3. **public** **static** < E > **void** printArray(E[] elements) {
4. **for** ( E element : elements){
5. System.out.println(element );
6. }
7. System.out.println();
8. }
9. **public** **static** **void** main( String args[] ) {
10. Integer[] intArray = { 10, 20, 30, 40, 50 };
11. Character[] charArray = { 'J', 'A', 'V', 'A', 'T','P','O','I','N','T' };
13. System.out.println( "Printing Integer Array" );
14. printArray( intArray  );
16. System.out.println( "Printing Character Array" );
17. printArray( charArray );
18. }
19. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestGenerics4)

**Output**

Printing Integer Array

10

20

30

40

50

Printing Character Array

J

A

V

A

T

P

O

I

N

T

## Wildcard in Java Generics

The ? (question mark) symbol represents the wildcard element. It means any type. If we write <? extends Number>, it means any child class of Number, e.g., Integer, Float, and double. Now we can call the method of Number class through any child class object.

We can use a wildcard as a **type of a parameter, field, return type, or local variable. However, it is not allowed to use a wildcard as a type argument for a generic method invocation, a generic class instance creation, or a supertype**.

Let's understand it by the example given below:

1. **import** java.util.\*;
2. **abstract** **class** Shape{
3. **abstract** **void** draw();
4. }
5. **class** Rectangle **extends** Shape{
6. **void** draw(){System.out.println("drawing rectangle");}
7. }
8. **class** Circle **extends** Shape{
9. **void** draw(){System.out.println("drawing circle");}
10. }
11. **class** GenericTest{
12. //creating a method that accepts only child class of Shape
13. **public** **static** **void** drawShapes(List<? **extends** Shape> lists){
14. **for**(Shape s:lists){
15. s.draw();//calling method of Shape class by child class instance
16. }
17. }
18. **public** **static** **void** main(String args[]){
19. List<Rectangle> list1=**new** ArrayList<Rectangle>();
20. list1.add(**new** Rectangle());
22. List<Circle> list2=**new** ArrayList<Circle>();
23. list2.add(**new** Circle());
24. list2.add(**new** Circle());
26. drawShapes(list1);
27. drawShapes(list2);
28. }}

**Output**

drawing rectangle

drawing circle

drawing circle

### **Upper Bounded Wildcards**

The purpose of upper bounded wildcards is to decrease the restrictions on a variable. It restricts the unknown type to be a specific type or a subtype of that type. It is used by declaring wildcard character ("?") followed by the extends (in case of, class) or implements (in case of, interface) keyword, followed by its upper bound.

### **Syntax**

1. List<? **extends** Number>

Here,

**?** is a wildcard character.

**extends**, is a keyword.

**Number**, is a class present in java.lang package

Suppose, we want to write the method for the list of Number and its subtypes (like Integer, Double). Using **List<? extends Number>** is suitable for a list of type Number or any of its subclasses whereas **List<Number>** works with the list of type Number only. So, **List<? extends Number>** is less restrictive than **List<Number>**.

### **Example of Upper Bound Wildcard**

In this example, we are using the upper bound wildcards to write the method for List<Integer> and List<Double>.

1. **import** java.util.ArrayList;
3. **public** **class** UpperBoundWildcard {

6. **private** **static** Double add(ArrayList<? **extends** Number> num) {
8. **double** sum=0.0;
10. **for**(Number n:num)
11. {
12. sum = sum+n.doubleValue();
13. }
15. **return** sum;
16. }
18. **public** **static** **void** main(String[] args) {
20. ArrayList<Integer> l1=**new** ArrayList<Integer>();
21. l1.add(10);
22. l1.add(20);
23. System.out.println("displaying the sum= "+add(l1));
25. ArrayList<Double> l2=**new** ArrayList<Double>();
26. l2.add(30.0);
27. l2.add(40.0);
28. System.out.println("displaying the sum= "+add(l2));

31. }
33. }

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

**Output**

displaying the sum= 30.0

displaying the sum= 70.0

### **Unbounded Wildcards**

The unbounded wildcard type represents the list of an unknown type such as List<?>. This approach can be useful in the following scenarios: -

* When the given method is implemented by using the functionality provided in the Object class.
* When the generic class contains the methods that don't depend on the type parameter.

### **Example of Unbounded Wildcards**

1. **import** java.util.Arrays;
2. **import** java.util.List;
4. **public** **class** UnboundedWildcard {
6. **public** **static** **void** display(List<?> list)
7. {
9. **for**(Object o:list)
10. {
11. System.out.println(o);
12. }
14. }

17. **public** **static** **void** main(String[] args) {
19. List<Integer> l1=Arrays.asList(1,2,3);
20. System.out.println("displaying the Integer values");
21. display(l1);
22. List<String> l2=Arrays.asList("One","Two","Three");
23. System.out.println("displaying the String values");
24. display(l2);
25. }
27. }

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

**Output**

displaying the Integer values

1

2

3

displaying the String values

One

Two

Three

## Lower Bounded Wildcards

The purpose of lower bounded wildcards is to restrict the unknown type to be a specific type or a supertype of that type. It is used by declaring wildcard character ("?") followed by the super keyword, followed by its lower bound.

### **Syntax**

1. List<? **super** Integer>

Here,

**?** is a wildcard character.

**super**, is a keyword.

**Integer**, is a wrapper class.

Suppose, we want to write the method for the list of Integer and its supertype (like Number, Object). Using **List<? super Integer>** is suitable for a list of type Integer or any of its superclasses whereas **List<Integer>** works with the list of type Integer only. So, **List<? super Integer>** is less restrictive than **List<Integer>**.

### **Example of Lower Bound Wildcard**

In this example, we are using the lower bound wildcards to write the method for List<Integer> and List<Number>.

1. **import** java.util.Arrays;
2. **import** java.util.List;
4. **public** **class** LowerBoundWildcard {
6. **public** **static** **void** addNumbers(List<? **super** Integer> list) {
8. **for**(Object n:list)
9. {
10. System.out.println(n);
11. }


15. }
16. **public** **static** **void** main(String[] args) {
18. List<Integer> l1=Arrays.asList(1,2,3);
19. System.out.println("displaying the Integer values");
20. addNumbers(l1);
22. List<Number> l2=Arrays.asList(1.0,2.0,3.0);
23. System.out.println("displaying the Number values");
24. addNumbers(l2);
25. }
27. }

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

**Output**

displaying the Integer values

1

2

3

displaying the Number values

1.0

2.0

3.0

StreamAPI:

A Stream represents a sequence of elements supporting sequential and parallel aggregate operations. Stream does not store data, it operates on source data structures such as List, [Collection](https://java2blog.com/collections-java/), Array etc.

Stream provides following features:

* Stream does not store elements. It simply conveys elements from a source such as a data structure, an array, or an I/O channel, through a pipeline of computational operations.
* Stream is functional in nature. Operations performed on a stream does not modify it's source. For example, filtering a Stream obtained from a collection produces a new Stream without the filtered elements, rather than removing elements from the source collection.
* Stream is lazy and evaluates code only when required.

The elements of a stream are only visited once during the life of a stream. Like an Iterator, a new stream must be generated to revisit the same elements of the source. **package** com.demo;

**public** **class** Employee {

**private** String empId;

**private** String firstName;

**private** String lastName;

**private** String email;

**private** String gender;

**private** String newJoiner;

**private** **int** salary;

**private** **int** rating;

**public** Employee(String empId, String firstName, String lastName, String email, String gender, String newJoiner,

**int** salary, **int** rating) {

**super**();

**this**.empId = empId;

**this**.firstName = firstName;

**this**.lastName = lastName;

**this**.email = email;

**this**.gender = gender;

**this**.newJoiner = newJoiner;

**this**.salary = salary;

**this**.rating = rating;

}

**public** Employee() {

**super**();

// **TODO** Auto-generated constructor stub

}

**public** String getEmpId() {

**return** empId;

}

**public** **void** setEmpId(String empId) {

**this**.empId = empId;

}

**public** String getFirstName() {

**return** firstName;

}

**public** **void** setFirstName(String firstName) {

**this**.firstName = firstName;

}

**public** String getLastName() {

**return** lastName;

}

**public** **void** setLastName(String lastName) {

**this**.lastName = lastName;

}

**public** String getEmail() {

**return** email;

}

**public** **void** setEmail(String email) {

**this**.email = email;

}

**public** String getGender() {

**return** gender;

}

**public** **void** setGender(String gender) {

**this**.gender = gender;

}

**public** String getNewJoiner() {

**return** newJoiner;

}

**public** **void** setNewJoiner(String newJoiner) {

**this**.newJoiner = newJoiner;

}

**public** **int** getSalary() {

**return** salary;

}

**public** **void** setSalary(**int** salary) {

**this**.salary = salary;

}

**public** **int** getRating() {

**return** rating;

}

**public** **void** setRating(**int** rating) {

**this**.rating = rating;

}

@Override

**public** String toString() {

**return** "Employee [empId=" + empId + ", firstName=" + firstName + ", lastName=" + lastName + ", email=" + email

+ ", gender=" + gender + ", newJoiner=" + newJoiner + ", salary=" + salary + ", rating=" + rating + "]";

}

* }

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

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

List<Employee> empList = *getEmpList*();

//We will filter the employees list with gender as Female

empList.stream().filter(e -> e.getGender().equals("Female")).forEach(e -> System.***out***.println(e));

//We will filter the employees list with newJoiner as True

empList.stream().filter(e -> e.getNewJoiner().equals("True")).forEach(e -> System.***out***.println(e));

// We will sort the employee list by rating asc.

empList.stream()

.sorted(Comparator.*comparing*(Employee::getRating))

.forEach(e -> System.***out***.println(e));

// We will sort the employee list by both rating and salary

empList.stream()

.sorted(Comparator.*comparing*(Employee::getSalary))

.sorted(Comparator.*comparing*(Employee::getRating))

.forEach(e -> System.***out***.println(e));

//We will see all employees with salary more than 1000

**boolean** isSalary = empList.stream()

.allMatch(e -> e.getSalary() > 1000);

System.***out***.println(isSalary);

//We will retrieve employee with max salary

empList.stream()

.max(Comparator.*comparing*(Employee::getSalary))

.ifPresent(System.***out***::println);

//We will retrieve employee with max rating

empList.stream()

.max(Comparator.*comparing*(Employee::getRating))

.ifPresent(System.***out***::println);

//We will retrieve employee with min salary

empList.stream()

.min(Comparator.*comparing*(Employee::getSalary))

.ifPresent(System.***out***::println);

// We will group all our employees by Gender

Map<String, List<Employee>> employeesBygender = empList.stream()

.collect(Collectors.*groupingBy*(Employee::getGender));

employeesBygender.forEach(((g,e)->{

System.***out***.println(g);

e.forEach(System.***out***::println);

}));

}

**public** **static** List<Employee> getEmpList(){

**return** Arrays.*asList*(

**new** Employee("59-385-1088","Zacharias","Schwerin","zchwerin@gmail.com","Male","True",101146,0),

**new** Employee("73-274-6476","Kyle","Frudd","kfrudd1@ovh.net","Male","FALSE",29310,2),

**new** Employee("85-939-9584","Axe","Gumb","agumb2@twitter.com","Female","FALSE",62291,4),

**new** Employee("08-180-8292","Robinet","Batterham","rbatterham3@last.fm","Male","FALSE",142439,4),

**new** Employee("21-825-2623","Ulick","Burrel","uburrel4@google.ru","Male","FALSE",128764,5),

**new** Employee("66-708-5539","Tailor","Ridding","Ridding","Female","FALSE",152924,4),

**new** Employee("81-697-2363","Joete","Braybrooke","jbraybrooke6@prnewswire.com","Male","TRUE",128907,0),

**new** Employee("63-019-1110","Elroy","Baverstock","ebaverstock7@ehow.com","Male","TRUE",2510,0)

);

}

}

Optional Class:

Every Java Programmer is familiar with [NullPointerException](https://www.geeksforgeeks.org/null-pointer-exception-in-java/). It can crash your code. And it is very hard to avoid it without using too many null checks. So, to overcome this, Java 8 has introduced a new class Optional in **java.util package**. It can help in writing a neat code without using too many null checks. By using Optional, we can specify alternate values to return or alternate code to run. This makes the code more readable because the facts which were hidden are now visible to the developer.

* Java

|  |
| --- |
| // Java program without Optional Class    **public** **class** OptionalDemo {  **public** **static** **void** main(String[] args)      {          String[] words = **new** String[10];          String word = words[5].toLowerCase();          System.out.print(word);      }  } |

**Output:**

Exception in thread "main" java.lang.NullPointerException

To avoid abnormal termination, we use the Optional class. In the following example, we are using Optional. So, our program can execute without crashing.

**The above program using Optional Class**

* Java

|  |
| --- |
| // Java program with Optional Class    **import** java.util.Optional;    **public** **class** OptionalDemo {  **public** **static** **void** main(String[] args)      {          String[] words = **new** String[10];          Optional<String> checkNull              = Optional.ofNullable(words[5]);  **if** (checkNull.isPresent()) {              String word = words[5].toLowerCase();              System.out.print(word);          }  **else**              System.out.println("word is null");      }  } |

**Output**

word is null

Optional is a container object which may or may not contain a non-null value. You must import ***java.util package*** to use this class. If a value is present, **isPresent()** will return true and **get()** will return the value. Additional methods that depend on the presence or absence of a contained value are provided, such as **orElse()** which returns a default value if the value is not present, and **ifPresent()** which executes a block of code if the value is present. This is a ***value-based*** class, i.e their instances are :

* Final and immutable (though may contain references to mutable objects).
* Considered equal solely based on equals(), not based on reference equality(==).
* Do not have accessible constructors.

**Static Methods:**Static methods are the methods in Java that can be called without creating an object of the class. They are referenced by the class name itself or reference to the object of that class.

**Syntax :**

public static void geek(String name)

{

// code to be executed....

}

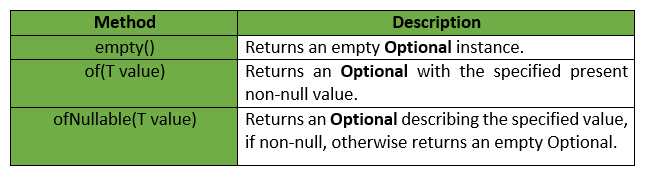
// Must have static modifier in their declaration.

// Return type can be int, float, String or user-defined data type.

**Important Points:** Since Static methods belong to the class, they can be called to without creating the object of the class. Below given are some important points regarding Static Methods :

* Static method(s) are associated with the class in which they reside i.e. they can be called even without creating an instance of the class.
* They are designed with the aim to be shared among all objects created from the same class.
* Static methods can not be overridden. But can be overloaded since they are resolved using static binding by the compiler at compile time.

The following table shows the list of Static Methods provided by Optional Class :



**Instance Methods:**Instance methods are methods that require an object of its class to be created before it can be called. To invoke an instance method, we have to create an Object of the class within which it is defined.

**Syntax :**

public void geek(String name)

{

// code to be executed....

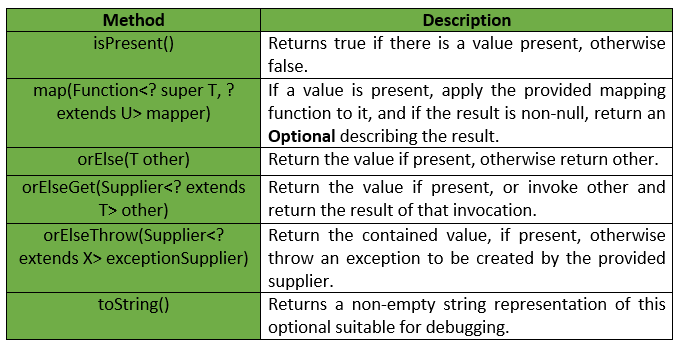
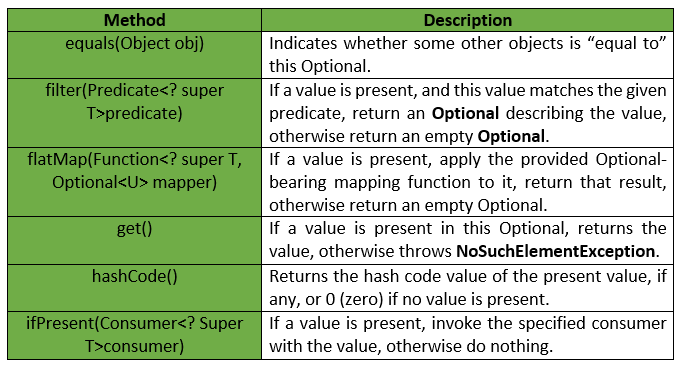
}

// Return type can be int, float String or user defined data type.

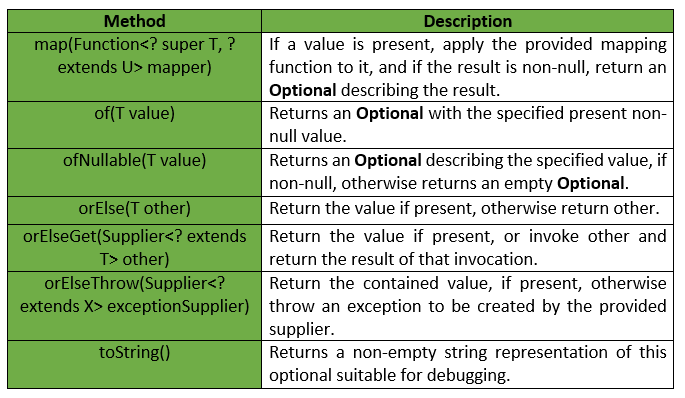
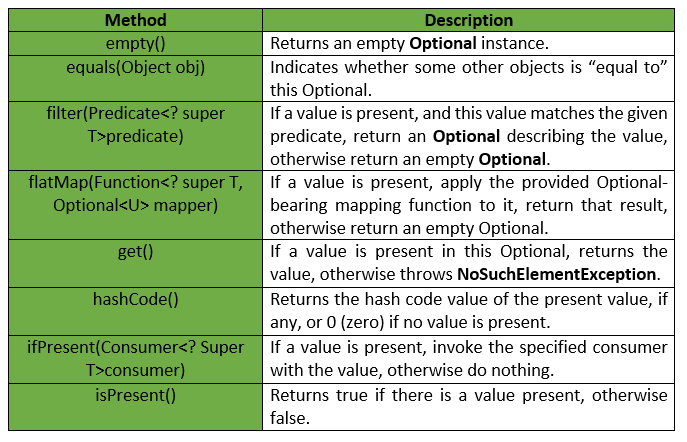
**Important Points:**Instance Methods can be called within the same class in which they reside or from the different classes defined either in the same package or other packages depending on the access type provided to the desired instance method. Below given are some important points regarding Instance Methods :

* Instance method(s) belong to the Object of the class, not to the class i.e. they can be called after creating the Object of the class.
* Every individual object created from the class has its own copy of the instance method(s) of that class.
* They can be overridden since they are resolved using dynamic binding at run time.

The following table shows the list of Instance Methods provided by the Optional Class :



**Concrete Methods:**A concrete method means, the method has a **complete definition** but can be overridden in the inherited class. If we make this method **final**, then it can not be overridden. Declaring method or class “final” means its implementation is complete. It is compulsory to override abstract methods. ***Concrete Methods can be overridden in the inherited classes if they are not final.*** The following table shows the list of Concrete Methods provided by the Optional Class :



Below given are some examples :

**Example 1 :**

* Java

|  |
| --- |
| // Java program to illustrate  // optional class methods    **import** java.util.Optional;    **class** GFG {        // Driver code  **public** **static** **void** main(String[] args)      {            // creating a string array          String[] str = **new** String[5];            // Setting value for 2nd index          str[2] = "Geeks Classes are coming soon";            // It returns an empty instance of Optional class          Optional<String> empty = Optional.empty();          System.out.println(empty);            // It returns a non-empty Optional          Optional<String> value = Optional.of(str[2]);          System.out.println(value);      }  } |

**Output**

Optional.empty

Optional[Geeks Classes are coming soon]

**Example 2 :**

* Java

|  |
| --- |
| // Java program to illustrate  // optional class methods    **import** java.util.Optional;    **class** GFG {        // Driver code  **public** **static** **void** main(String[] args)      {            // creating a string array          String[] str = **new** String[5];            // Setting value for 2nd index          str[2] = "Geeks Classes are coming soon";            // It returns a non-empty Optional          Optional<String> value = Optional.of(str[2]);            // It returns value of an Optional.          // If value is not present, it throws          // an NoSuchElementException          System.out.println(value.get());            // It returns hashCode of the value          System.out.println(value.hashCode());            // It returns true if value is present,          // otherwise false          System.out.println(value.isPresent());      }  } |

**Output**

Geeks Classes are coming soon

1967487235

True

### **Example: Java Program without using Optional**

In the following example, we are not using Optional class. This program terminates abnormally and throws a nullPointerException.

1. **public** **class** OptionalExample {
2. **public** **static** **void** main(String[] args) {
3. String[] str = **new** String[10];
4. String lowercaseString = str[5].toLowerCase();
5. System.out.print(lowercaseString);
6. }
7. }

Output:

Exception in thread "main" java.lang.NullPointerException

at lambdaExample.OptionalExample.main(OptionalExample.java:6)

To avoid the abnormal termination, we use Optional class. In the following example, we are using Optional. So, our program can execute without crashing.

57.6M

1.1K

Prime Ministers of India | List of Prime Minister of India (1947-2020)

### **Java Optional Example: If Value is not Present**

1. **import** java.util.Optional;
2. **public** **class** OptionalExample {
3. **public** **static** **void** main(String[] args) {
4. String[] str = **new** String[10];
5. Optional<String> checkNull = Optional.ofNullable(str[5]);
6. **if**(checkNull.isPresent()){  // check for value is present or not
7. String lowercaseString = str[5].toLowerCase();
8. System.out.print(lowercaseString);
9. }**else**
10. System.out.println("string value is not present");
11. }
12. }

Output:

string value is not present

### **Java Optional Example: If Value is Present**

1. **import** java.util.Optional;
2. **public** **class** OptionalExample {
3. **public** **static** **void** main(String[] args) {
4. String[] str = **new** String[10];
5. str[5] = "JAVA OPTIONAL CLASS EXAMPLE";// Setting value for 5th index
6. Optional<String> checkNull = Optional.ofNullable(str[5]);
7. **if**(checkNull.isPresent()){  // It Checks, value is present or not
8. String lowercaseString = str[5].toLowerCase();
9. System.out.print(lowercaseString);
10. }**else**
11. System.out.println("String value is not present");
12. }
13. }

Output:

java optional class example

### **Another Java Optional Example**

1. **import** java.util.Optional;
2. **public** **class** OptionalExample {
3. **public** **static** **void** main(String[] args) {
4. String[] str = **new** String[10];
5. str[5] = "JAVA OPTIONAL CLASS EXAMPLE";  // Setting value for 5th index
6. Optional<String> checkNull = Optional.ofNullable(str[5]);
7. checkNull.ifPresent(System.out::println);   // printing value by using method reference
8. System.out.println(checkNull.get());    // printing value by using get method
9. System.out.println(str[5].toLowerCase());
10. }
11. }

Output:

JAVA OPTIONAL CLASS EXAMPLE

JAVA OPTIONAL CLASS EXAMPLE

java optional class example

### **Java Optional Methods Example**

1. **import** java.util.Optional;
2. **public** **class** OptionalExample {
3. **public** **static** **void** main(String[] args) {
4. String[] str = **new** String[10];
5. str[5] = "JAVA OPTIONAL CLASS EXAMPLE";  // Setting value for 5th index
6. // It returns an empty instance of Optional class
7. Optional<String> empty = Optional.empty();
8. System.out.println(empty);
9. // It returns a non-empty Optional
10. Optional<String> value = Optional.of(str[5]);
11. // If value is present, it returns an Optional otherwise returns an empty Optional
12. System.out.println("Filtered value: "+value.filter((s)->s.equals("Abc")));
13. System.out.println("Filtered value: "+value.filter((s)->s.equals("JAVA OPTIONAL CLASS EXAMPLE")));
14. // It returns value of an Optional. if value is not present, it throws an NoSuchElementException
15. System.out.println("Getting value: "+value.get());
16. // It returns hashCode of the value
17. System.out.println("Getting hashCode: "+value.hashCode());
18. // It returns true if value is present, otherwise false
19. System.out.println("Is value present: "+value.isPresent());
20. // It returns non-empty Optional if value is present, otherwise returns an empty Optional
21. System.out.println("Nullable Optional: "+Optional.ofNullable(str[5]));
22. // It returns value if available, otherwise returns specified value,
23. System.out.println("orElse: "+value.orElse("Value is not present"));
24. System.out.println("orElse: "+empty.orElse("Value is not present"));
25. value.ifPresent(System.out::println);   // printing value by using method reference
26. }
27. }

Output:

Optional.empty

Filtered value: Optional.empty

Filtered value: Optional[JAVA OPTIONAL CLASS EXAMPLE]

Getting value: JAVA OPTIONAL CLASS EXAMPLE

Getting hashCode: -619947648

Is value present: true

Nullable Optional: Optional[JAVA OPTIONAL CLASS EXAMPLE]

orElse: JAVA OPTIONAL CLASS EXAMPLE

orElse: Value is not present

JAVA OPTIONAL CLASS EXAMPLE