**Lambda Expression :**

The Lambda expression is used to provide the implementation of 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.

**Why use Lambda Expression**

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

**Java Lambda Expression Syntax**

1. (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

## 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...");
26. // implementing lambda expression
27. Collections.sort(list, (p1,p2)->{
28. return p1.name.compareTo(p2.name);
29. });
30. for(Product p:list){
31. System.out.println(p.id+" "+p.name+" "+p.price);
32. }
34. }
35. }

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. }

Output:

Iphone 6S: 65000.0

Sony Xperia: 25000.0

Redmi4 : 26000.0

# Java Method References

Method reference is used to refer method of functional interface. It is compact and easy form of lambda expression. Each time when you are using lambda expression to just referring a method, you can replace your lambda expression with method reference.

**Types of Method References**

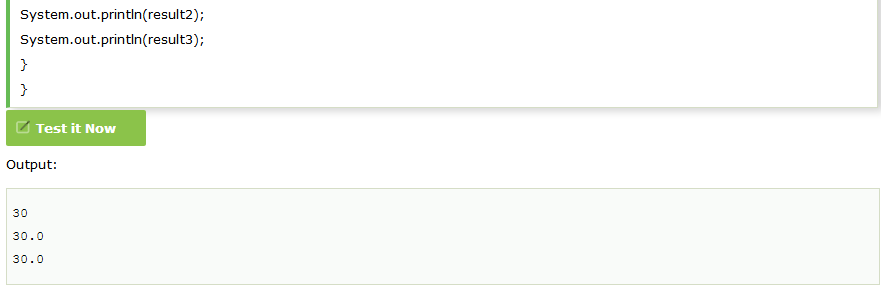
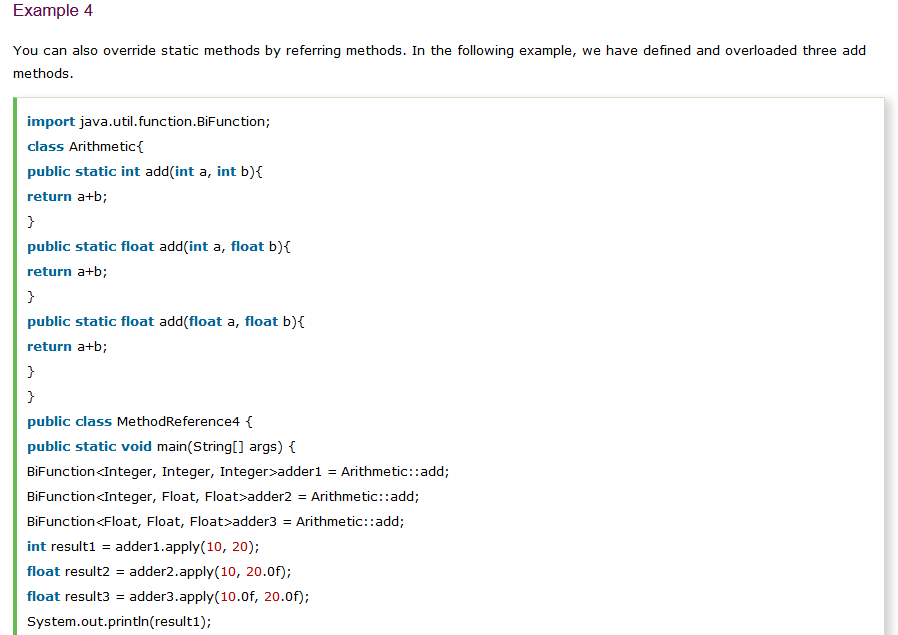
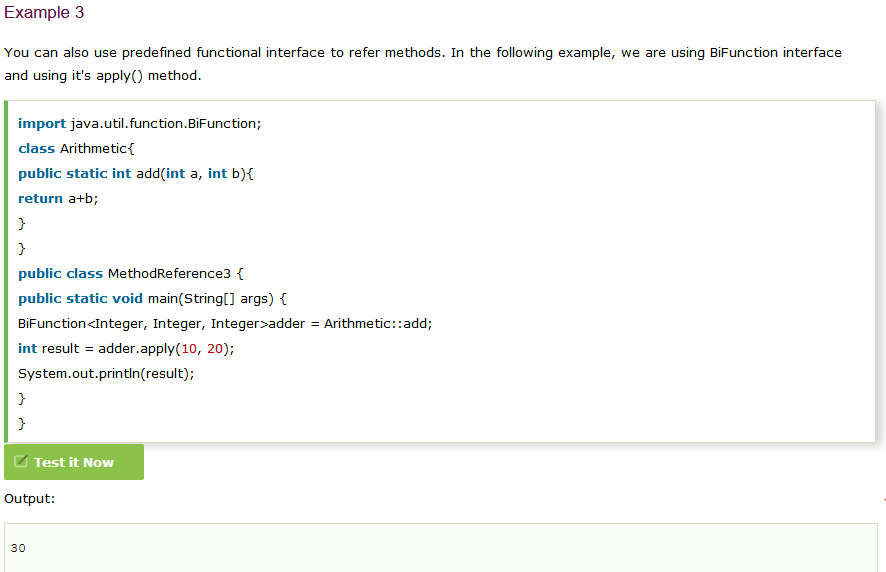
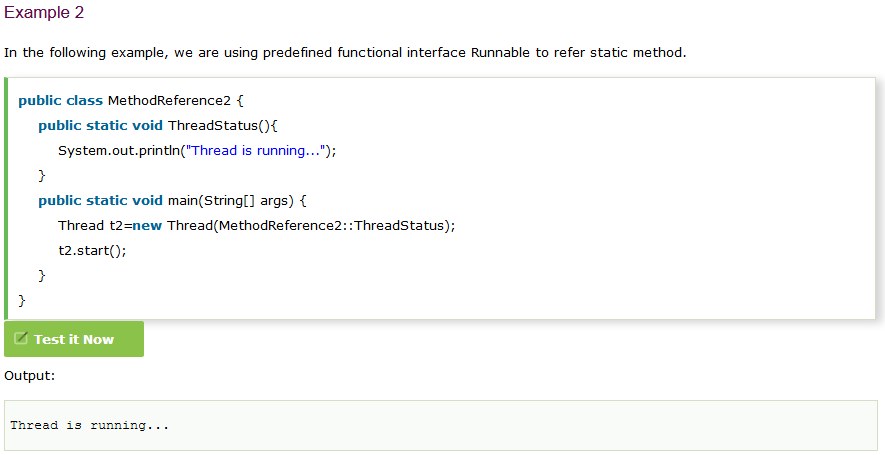
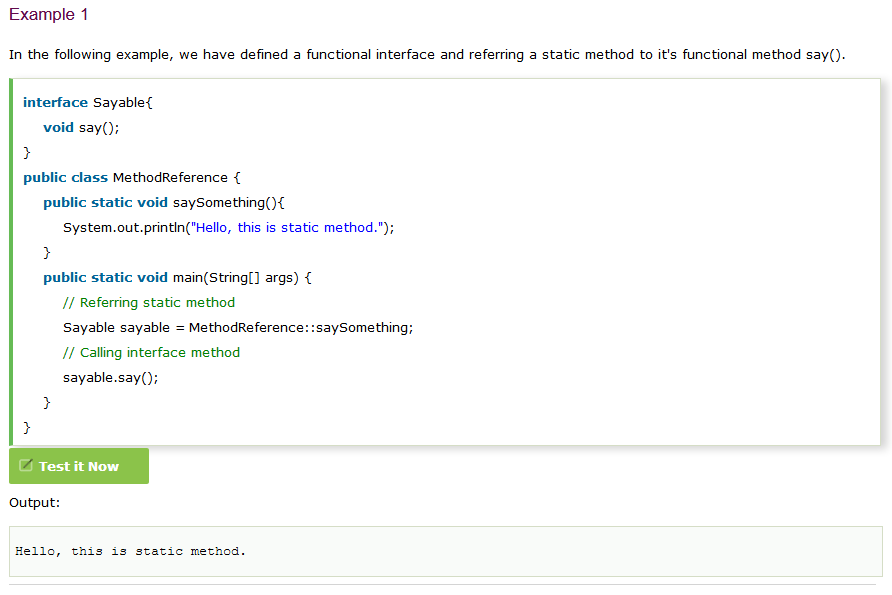
There are following types of method references in java:

1. Reference to a static method.
2. Reference to an instance method.
3. Reference to a constructor.

## 1) Reference to a Static Method

Syntax

**ContainingClass::staticMethodName**

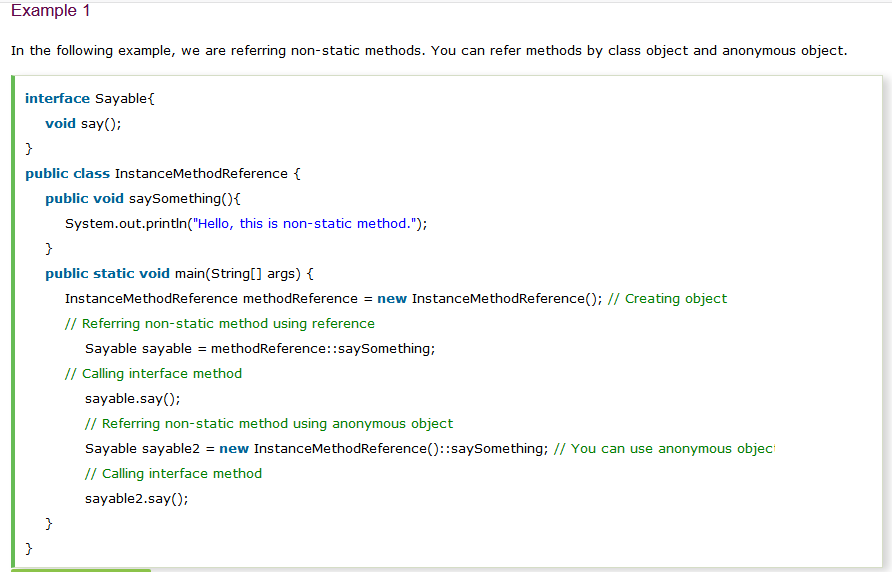


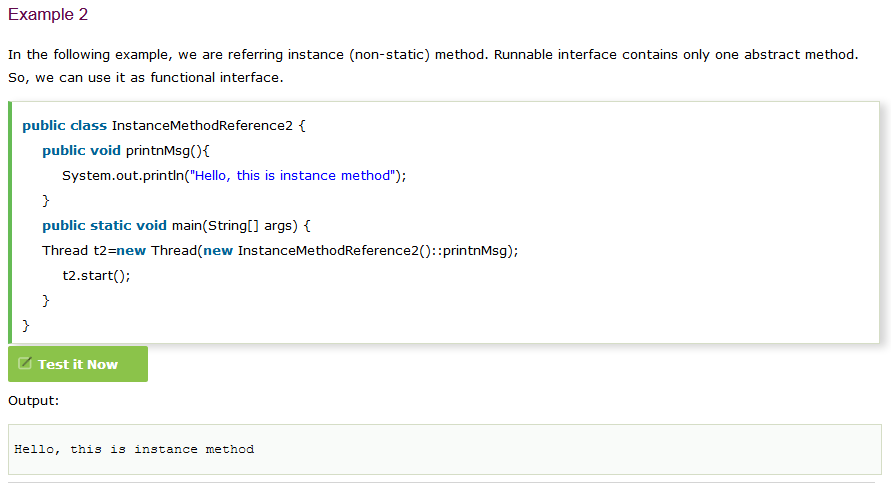
**2) Reference to an Instance Method**

like static methods, you can refer instance methods also. In the following example, we are describing the process of referring the instance method.

Syntax

**containingObject::instanceMethodName**



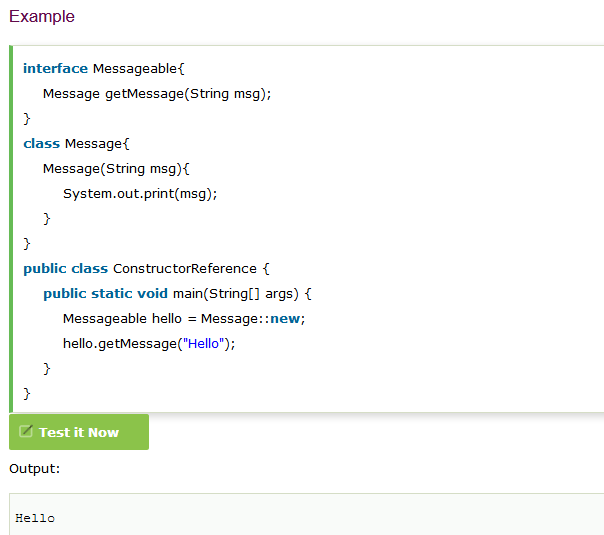


## 3) Reference to a Constructor

You can refer a constructor by using the new keyword. Here, we are referring constructor with the help of functional interface.

Syntax

**ClassName::new**



Java Functional Interfaces

An Interface that contains exactly one abstract method is known as functional interface. It can have any number of default, static methods but can contain only one abstract method. It can also declare methods of object class.

A functional interface can have methods of object class. See in the following example.

1. @FunctionalInterface
2. **interface** sayable{
3. **void** say(String msg);   // abstract method
4. // It can contain any number of Object class methods.
5. **int** hashCode();
6. String toString();
7. **boolean** equals(Object obj);
8. }
9. **public** **class** FunctionalInterfaceExample2 **implements** sayable{
10. **public** **void** say(String msg){
11. System.out.println(msg);
12. }
13. **public** **static** **void** main(String[] args) {
14. FunctionalInterfaceExample2 fie = **new** FunctionalInterfaceExample2();
15. fie.say("Hello there");
16. }
17. }

### Invalid Functional Interface

A functional interface can extends another interface only when it does not have any abstract method.

1. **interface** sayable{
2. **void** say(String msg);   // abstract method
3. }
4. @FunctionalInterface
5. **interface** Doable **extends** sayable{
6. // Invalid '@FunctionalInterface' annotation; Doable is not a functional interface
7. **void** doIt();
8. }

Output:

compile-time error

### Example 3

In the following example, a functional interface is extending to a non-functional interface.

1. interface Doable{
2. default void doIt(){
3. System.out.println("Do it now");
4. }
5. }
6. @FunctionalInterface
7. interface Sayable extends Doable{
8. void say(String msg);   // abstract method
9. }
10. public class FunctionalInterfaceExample3 implements Sayable{
11. public void say(String msg){
12. System.out.println(msg);
13. }
14. public static void main(String[] args) {
15. FunctionalInterfaceExample3 fie = new FunctionalInterfaceExample3();
16. fie.say("Hello there");
17. fie.doIt();
18. }
19. }

Output:

Hello there

Do it now

**Java 8 has some new Functional Interface:**

1. **Function Interface:** It is a functional interface. It is used to refer method by specifying type of parameter. It returns a result back to the referred function.

### Java Function Interface Example 1

1. // importing Function interface
2. import java.util.function.Function;
3. public class FunctionInterfaceExample {
4. static String show(String message){
5. return "Hello "+message;
6. }
7. public static void main(String[] args) {
8. // Function interface referring to a method
9. Function<String, String> fun = FunctionInterfaceExample::show;
10. // Calling Function interface method
11. System.out.println(fun.apply("Peter"));
12. }
13. }

**Output:**

**Hello Peter**

### Java Function Interface Example 2

1. // importing Function interface
2. import java.util.function.Function;
3. import java.util.List;
4. import java.util.ArrayList;
5. public class FunctionInterfaceExample {
6. static Integer addList(List<Integer> list){
7. return list.stream()
8. .mapToInt(Integer::intValue)
9. .sum();
10. }
11. public static void main(String[] args) {
12. // Creating a list and adding values
13. List<Integer> list = new ArrayList<Integer>();
14. list.add(10);
15. list.add(20);
16. list.add(30);
17. list.add(40);
18. // Referring addList() method
19. Function<List<Integer>, Integer> fun = FunctionInterfaceExample::addList;
20. // Calling Function interface method
21. int result = fun.apply(list);
22. System.out.println("Sum of list values: "+result);
23. }
24. }

Output:

Sum of list values: 100

# Java Predicate Interface

It is a functional interface which represents a predicate (boolean-valued function) of one argument. It is defined in the java.util.function package and contains test() a functional method.

### Java Predicate Interface Example 1

1. import java.util.function.Predicate;
2. public class PredicateInterfaceExample {
3. public static void main(String[] args) {
4. Predicate<Integer> pr = a -> (a > 18); // Creating predicate
5. System.out.println(pr.test(10));    // Calling Predicate method
6. }
7. }

Output:

false

### Java Predicate Interface Example 2

1. import java.util.function.Predicate;
2. public class PredicateInterfaceExample {
3. static Boolean checkAge(int age){
4. if(age>17)
5. return true;
6. else return false;
7. }
8. public static void main(String[] args){
9. // Using Predicate interface
10. Predicate<Integer> predicate =  PredicateInterfaceExample::checkAge;
11. // Calling Predicate method
12. boolean result = predicate.test(25);
13. System.out.println(result);
14. }
15. }

Output:

true

# Java Consumer Interface

It is a functional interface defined in java.util.function package. It contains an abstract accept() and a default andThen() method. It can be used as the assignment target for a lambda expression or method reference.

The Consumer Interface accepts a single argument and does not return any result.

### Java Consumer Interface Example 1

1. // Importing Consumer interface
2. import java.util.function.Consumer;
3. public class ConsumerInterfaceExample {
4. static void printMessage(String name){
5. System.out.println("Hello "+name);
6. }
7. static void printValue(int val){
8. System.out.println(val);
9. }
10. public static void main(String[] args) {
11. // Referring method to String type Consumer interface
12. Consumer<String> consumer1 = ConsumerInterfaceExample::printMessage;
13. consumer1.accept("John");   // Calling Consumer method
14. // Referring method to Integer type Consumer interface
15. Consumer<Integer> consumer2 = ConsumerInterfaceExample::printValue;
16. consumer2.accept(12);   // Calling Consumer method
17. }
18. }

Output:

Hello John

12

### Java Consumer Interface Example2

1. import java.util.ArrayList;
2. import java.util.List;
3. import java.util.function.Consumer;
4. public class ConsumerInterfaceExample {
5. static void addList(List<Integer> list){
6. // Return sum of list values
7. int result = list.stream()
8. .mapToInt(Integer::intValue)
9. .sum();
10. System.out.println("Sum of list values: "+result);
11. }
12. public static void main(String[] args) {
13. // Creating a list and adding values
14. List<Integer> list = new ArrayList<Integer>();
15. list.add(10);
16. list.add(20);
17. list.add(30);
18. list.add(40);
19. // Referring method to String type Consumer interface
20. Consumer<List<Integer>> consumer = ConsumerInterfaceExample::addList;
21. consumer.accept(list);  // Calling Consumer method
23. }
24. }

Output:

Sum of list values: 100

# Java BiConsumer Interface

BiConsumer Interface accepts two input arguments and does not return any result.

### Java BiConsumer Interface Example 1

1. import java.util.function.BiConsumer;
2. public class BiConsumerInterfaceExample {
3. static void ShowDetails(String name, Integer age){
4. System.out.println(name+" "+age);
5. }
6. public static void main(String[] args) {
7. // Referring method
8. BiConsumer<String, Integer> biCon = BiConsumerInterfaceExample::ShowDetails;
9. biCon.accept("Rama", 20);
10. biCon.accept("Shyam", 25);
11. // Using lambda expression
12. BiConsumer<String, Integer> biCon2 = (name, age)->System.out.println(name+" "+age);
13. biCon2.accept("Peter", 28);
14. }
15. }

Output:

Rama 20

Shyam 25

Peter 28

### Java BiConsumer Interface Example 2

1. import java.util.function.BiConsumer;
2. import java.util.HashMap;
3. import java.util.Map;
4. public class BiConsumerInterfaceExample {
5. static void ShowDetails(Map<Integer, String> map, String mapName){
6. System.out.println("----------"+mapName+" records-----------");
7. map.forEach((key, val)->System.out.println(key+" "+val));
8. }
9. public static void main(String[] args) {
10. Map<Integer, String> map = new HashMap<Integer,String>();
11. map.put(100, "Mohan");
12. map.put(110, "Sujeet");
13. map.put(115, "Tom");
14. map.put(120, "Danish");
15. // Referring method
16. BiConsumer<Map<Integer, String>, String> biCon = BiConsumerInterfaceExample::ShowDetails;
17. biCon.accept(map, "Student");
18. }
19. }

Output:

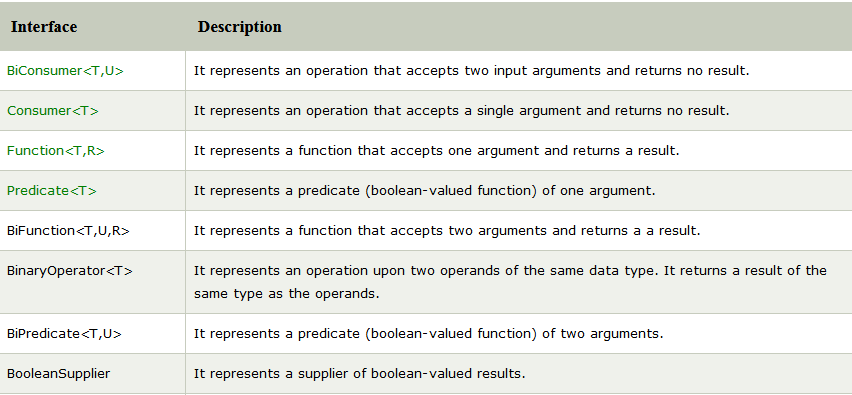
----------Student records-----------

115 Tom

100 Mohan

120 Danish

110 Sujeet



# Java 8 Stream

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.

You can use stream to filter, collect, print, and convert from one data structure to other etc. In the following examples, we have apply various operations with the help of stream.

### Java Example: Filtering Collection without using Stream

In the following example, we are filtering data without using stream. This approach we are used before the stream package was released.

1. import java.util.\*;
2. class Product{
3. int id;
4. String name;
5. float price;
6. public Product(int id, String name, float price) {
7. this.id = id;
8. this.name = name;
9. this.price = price;
10. }
11. }
12. public class JavaStreamExample {
13. public static void main(String[] args) {
14. List<Product> productsList = new ArrayList<Product>();
15. //Adding Products
16. productsList.add(new Product(1,"HP Laptop",25000f));
17. productsList.add(new Product(2,"Dell Laptop",30000f));
18. productsList.add(new Product(3,"Lenevo Laptop",28000f));
19. productsList.add(new Product(4,"Sony Laptop",28000f));
20. productsList.add(new Product(5,"Apple Laptop",90000f));
21. List<Float> productPriceList = new ArrayList<Float>();
22. for(Product product: productsList){
24. // filtering data of list
25. if(product.price<30000){
26. productPriceList.add(product.price);    // adding price to a productPriceList
27. }
28. }
29. System.out.println(productPriceList);   // displaying data
30. }
31. }

Output:

[25000.0, 28000.0, 28000.0]

### Java Stream Example: Filtering Collection by using Stream

Here, we are filtering data by using stream. You can see that code is optimized and maintained. Stream provides fast execution.

1. import java.util.\*;
2. import java.util.stream.Collectors;
3. class Product{
4. int id;
5. String name;
6. float price;
7. public Product(int id, String name, float price) {
8. this.id = id;
9. this.name = name;
10. this.price = price;
11. }
12. }
13. public class JavaStreamExample {
14. public static void main(String[] args) {
15. List<Product> productsList = new ArrayList<Product>();
16. //Adding Products
17. productsList.add(new Product(1,"HP Laptop",25000f));
18. productsList.add(new Product(2,"Dell Laptop",30000f));
19. productsList.add(new Product(3,"Lenevo Laptop",28000f));
20. productsList.add(new Product(4,"Sony Laptop",28000f));
21. productsList.add(new Product(5,"Apple Laptop",90000f));
22. List<Float> productPriceList2 =productsList.stream()
23. .filter(p -> p.price > 30000) // filtering data
24. .map(p->p.price)        // fetching price
25. .collect(Collectors.toList()); // collecting as list
26. System.out.println(productPriceList2);
27. }
28. }

Output:

[90000.0]

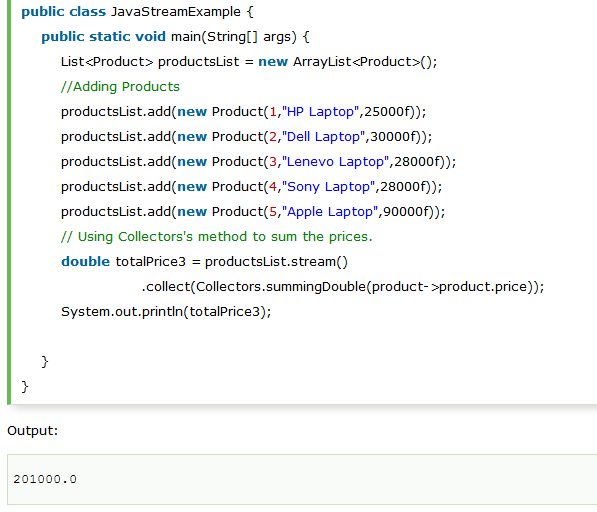
### Java Stream Example: Filtering and Iterating Collection

In the following example, we are using filter() method. Here, you can see code is optimized and very concise

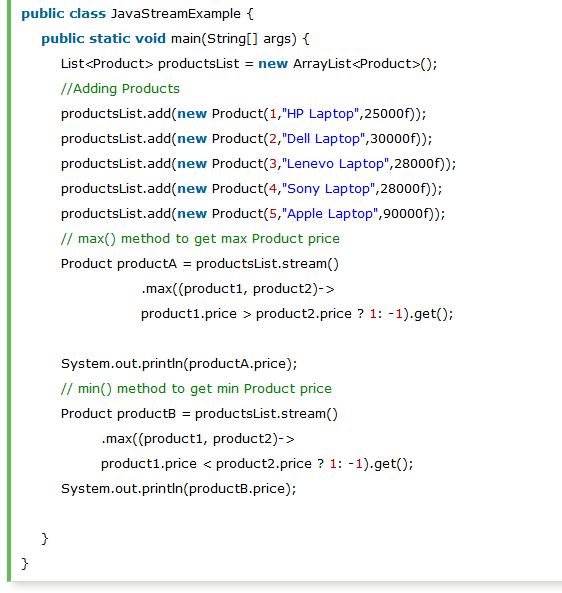
Output:

Dell Laptop

### Java Stream Example: Sum by using Collectors Methods



### Java Stream Example: Find Max and Min Product Price

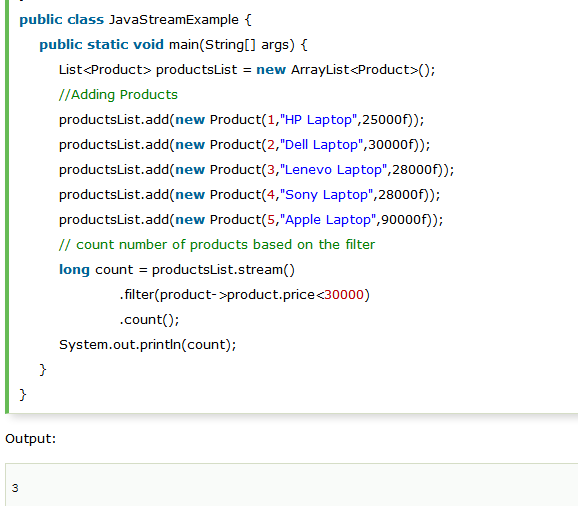


Output:

90000.0

25000.0

### Java Stream Example: count() Method in Collection

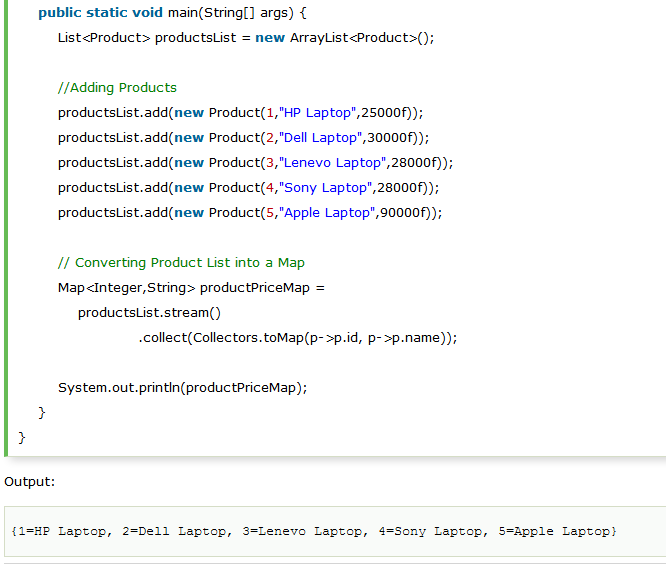


Note : Stream allows you to collect your result in any various forms. You can get you result as set, list or map and can perform manipulation on the elements.

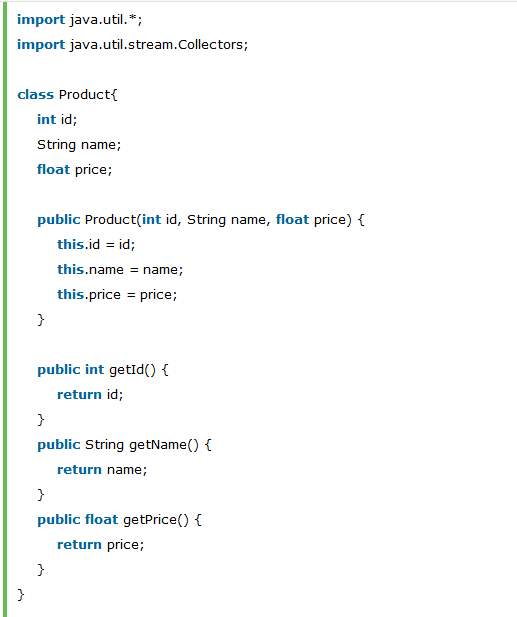
### Java Stream Example : Convert List into Set

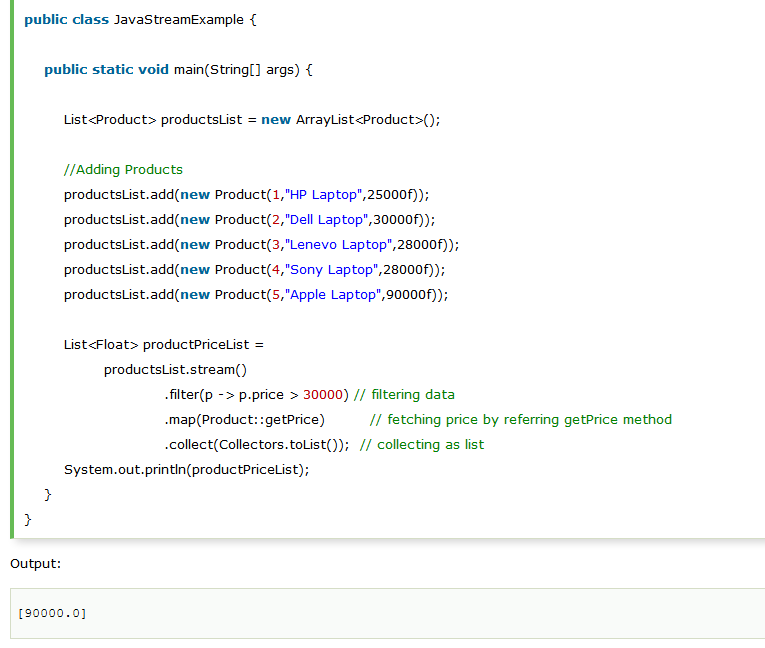


### Java Stream Example : Convert List into Map

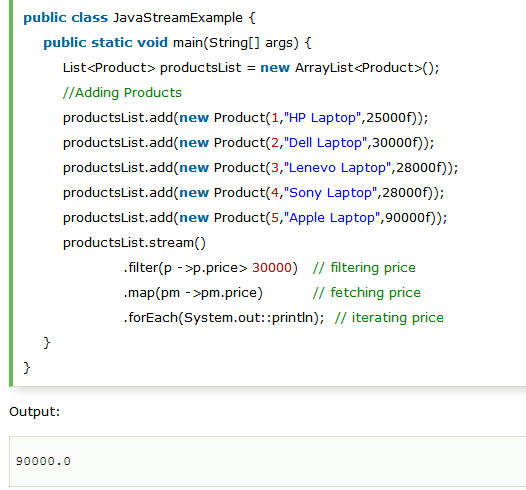


### Method Reference in stream

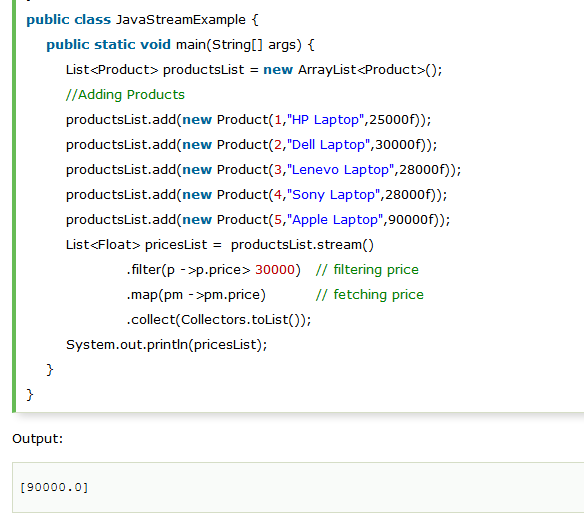




### Java Stream filter() example

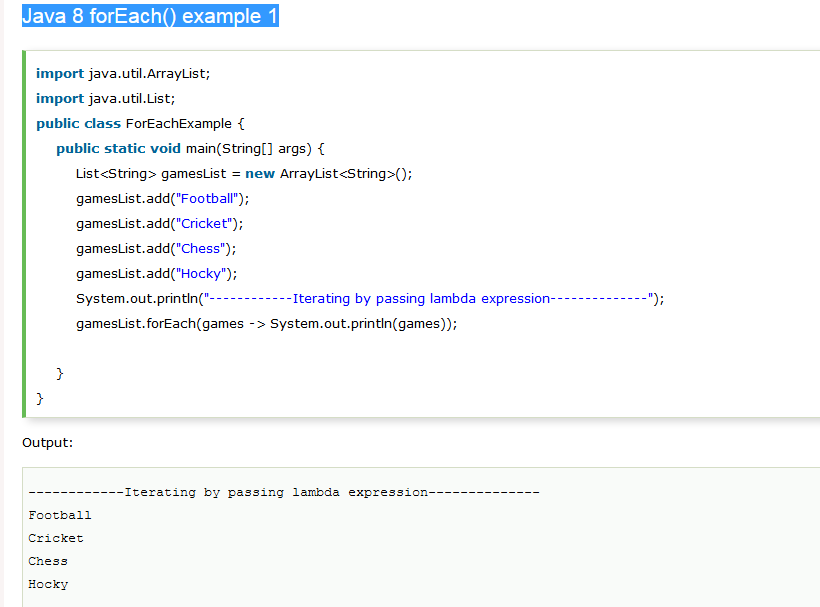


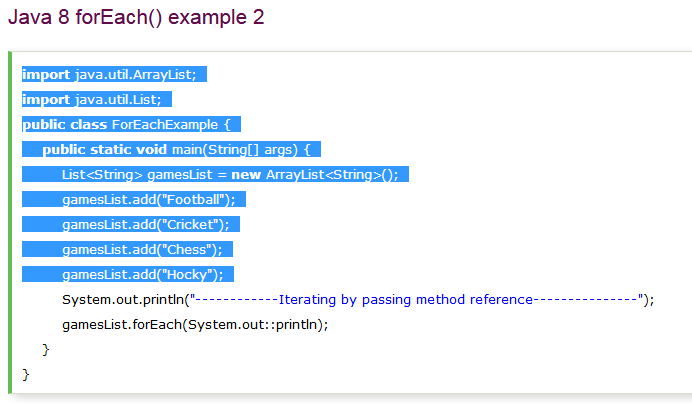
### Java Stream filter() example 2



# Java forEach loop:

Java provides a new method forEach() to iterate the elements. It is defined in Iterable and Stream interface. It is a default method defined in the Iterable interface.

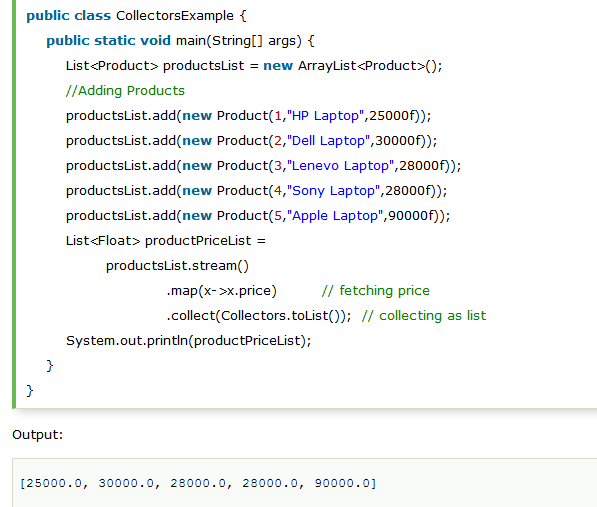




# Java Collectors:

Collectors is a final class that extends Object class. It provides reduction operations, such as accumulating elements into collections, summarizing elements according to various criteria, etc.

### Java Collectors Example: Fetching data as a List



**Converting data into a set :**

Set<Float>productPriceList =

                productsList.stream()

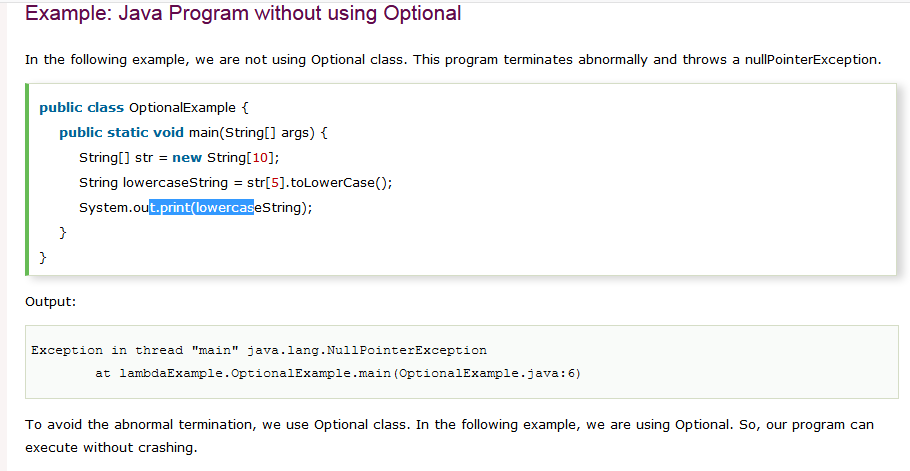
                            .map(x->x.price)         // fetching price

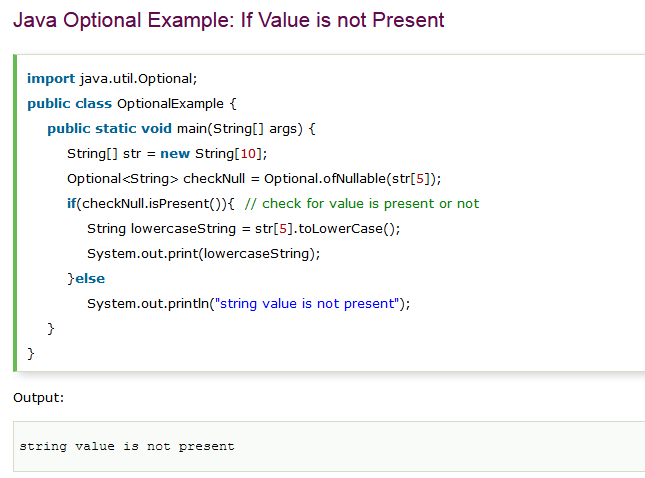
                            .collect(Collectors.toSet());   // collecting as list

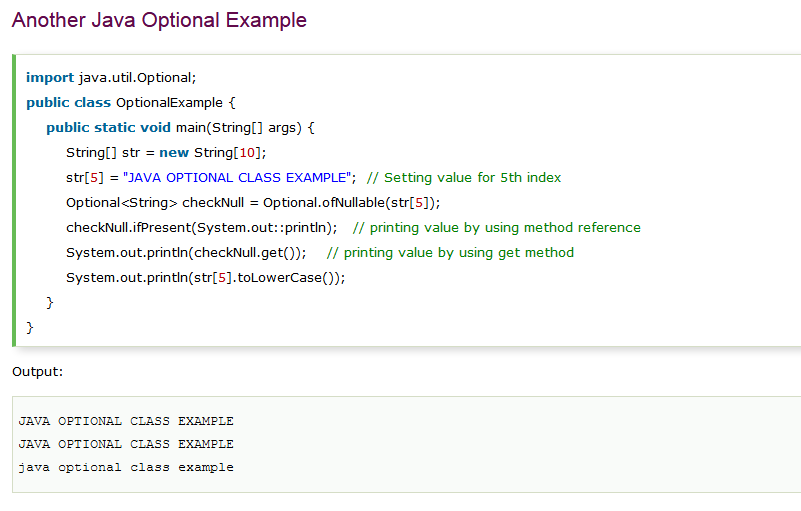
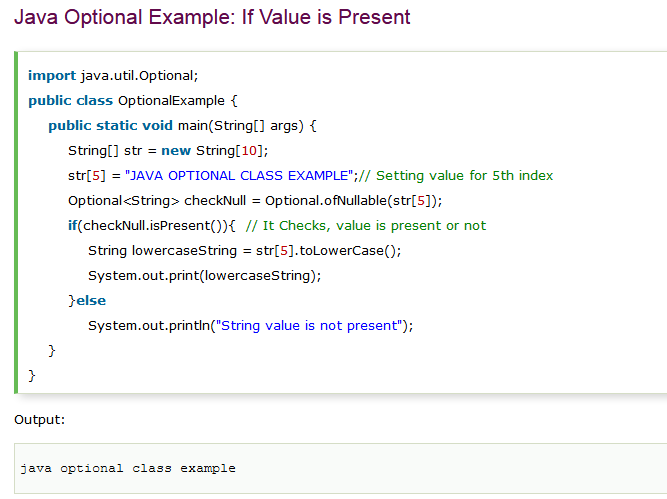
        System.out.println(productPriceList);

# Java Optional Class

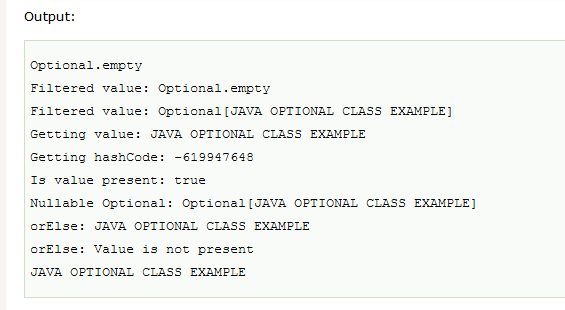
### It is a public final class and used to deal with NullPointerException in Java application. You must import java.util package to use this class. It provides methods which are used to check the presence of value for particular variable.







### Java Optional Methods Example



How to do in Java Optional:

**NullPointerException** happen when you try to utilize an object reference which has not been initialized, initialized with null or simply does not point to any instance. ***NULL simply means ‘absence of a value’***.

## 1) What is the Type of Null?

null is used everytime where we don’t know or we don’t have a value to give to a reference.

## 2) What is wrong with just returning null?

Generally, the API designers put the descriptive java docs in APIs and mention there that API can return a null value, and in which case(s). Now, the problem is that the caller of the API might have missed reading the javadoc for any reason, and **forget about handling the null case**. This is going to be a bug in future for sure.

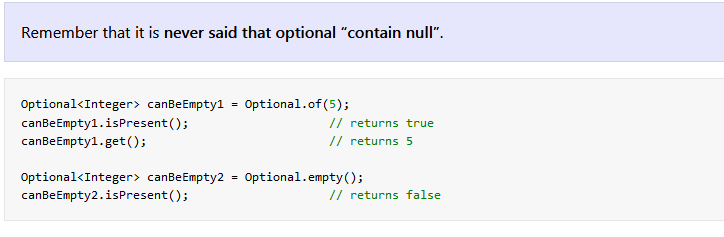
And believe me, this happens frequently and is one of the main causes of null pointer exception. A good solution is to always **initialize your object references with some value**, and never with null. In this way, you will never encounter NullPointerException.

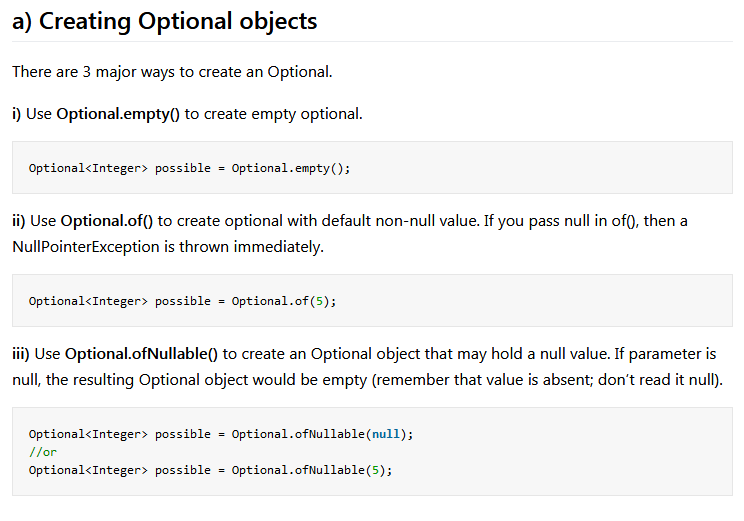
Fair enough. But in practical we always don’t have a default value for a reference. So, how those cases should be handled?

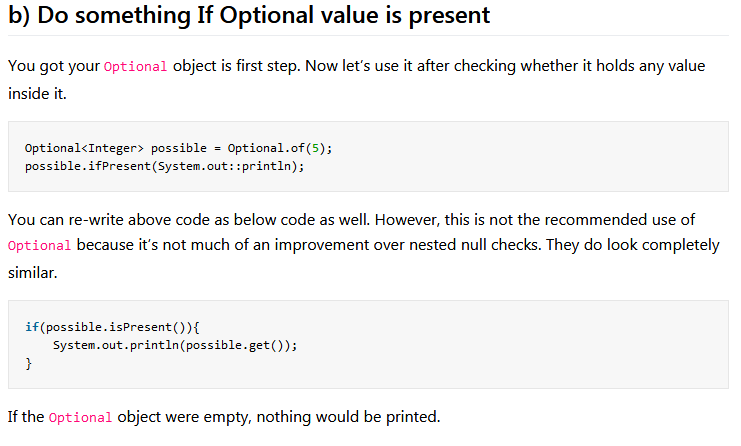
Above question is right in many senses. Well, **java 8 Optionals** are the answer here.

## 3) How Java 8 Optionals provide the solution?

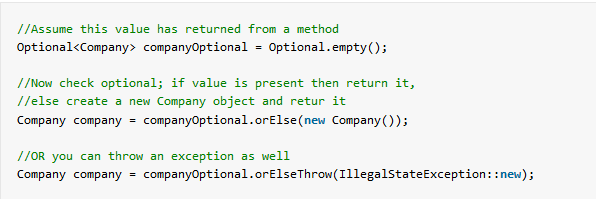
## Optional is a way of replacing a nullable T reference with a non-null value. An Optional may either contain a non-null T reference (in which case we say the reference is “present”), or it may contain nothing (in which case we say the reference is “absent”).

 You can also **view Optional as a single-value container that either contains a value or doesn’t**. It is important to note that the intention of the Optional class is not to replace every single null reference. Instead, its purpose is to **help design more-comprehensible APIs** so that by just reading the signature of a method, you can tell whether you can expect an optional value. This forces you to fetch the value from Optional and work on it.

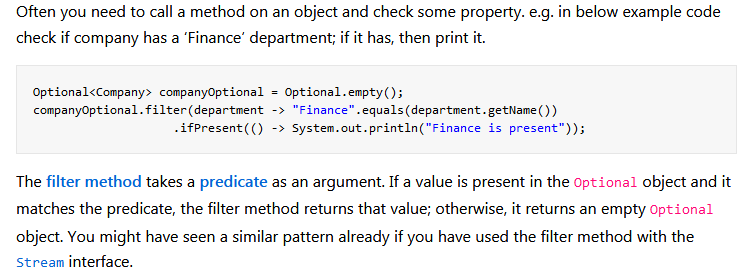




## c) Default/absent values and actions



## d) Rejecting certain values using the filter method



Note : When you try to **get a value from an Optional**, value is fetched if present other wise NoSuchElementException is thrown:

|  |
| --- |
| public T get() {      if (value == null) {          throw new NoSuchElementException("No value present");      }      return value;  } |

Note: **Optional** should be used almost all the time ***as the return type of functions*** that might not return a value.

## Conclusion

**java.util.Optional**. The purpose of **Optional** is not to replace every single null reference in your code base but rather to help you design better APIs in which, just by reading the signature of a method, users can tell whether to expect an optional value and deal with it appropriately.

----------------------------------------\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*-----------------------------------------------------