

**Lambda Expression**

* A lambda expression is an anonymous function. A function that doesn’t have a name and doesn’t belong to any class
* Lambdas are backward compatible so you can use them in existing API when you migrate your project to java 8.

## **Java Lambda Expression Syntax**

To create a lambda expression, we specify input parameters (if there are any) on the left side of the lambda operator ->, and place the expression or block of statements on the right side of lambda operator. For example, the lambda expression (x, y) -> x + y specifies that lambda expression takes two arguments x and y and returns the sum of these.

//Syntax of lambda expression

* (parameter\_list) -> {function\_body}

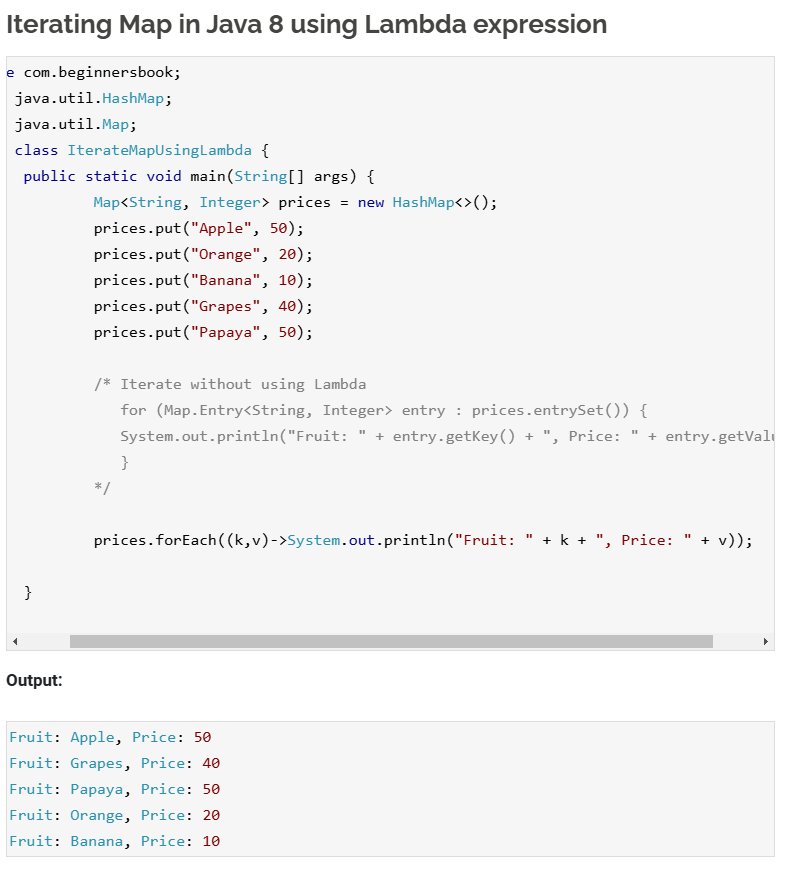
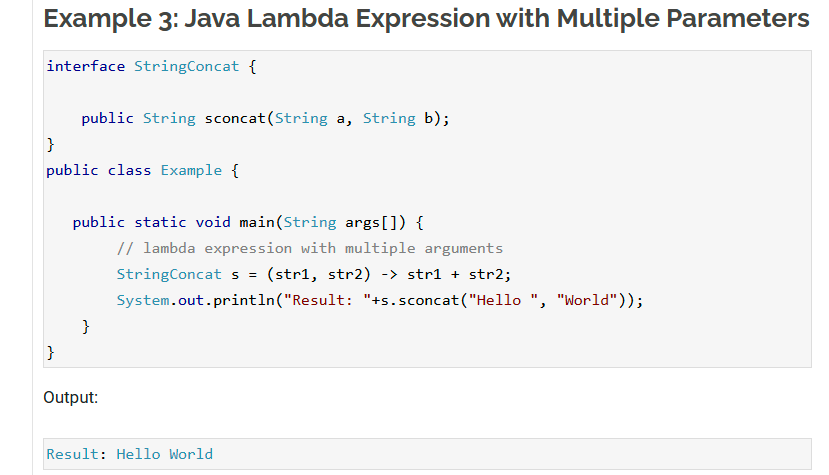
## **Lambda expression vs method in Java**

A method (or function) in Java has these main parts:  
1. Name  
2. Parameter list  
3. Body  
4. return type.

A lambda expression in Java has these main parts:  
Lambda expression **only has body and parameter list**.  
1. **No** name – function is anonymous so we don’t care about the name  
2. Parameter list  
3. Body – This is the main part of the function.  
4. **No** return type – The java 8 compiler is able to infer the return type by checking the code. you need not to mention it explicitly.

## **Where to use the Lambdas in Java**

To use lambda expression, you need to either create your own functional interface or use the pre defined functional interface provided by Java. An interface with **only single abstract method** is called functional interface(or Single Abstract method interface), for example: Runnable, callable, ActionListener etc.



# **Method References**

* Method reference is a shorthand notation of a lambda expression to call a method. For example:  
  If your lambda expression is like this:

str -> System.out.println(str)

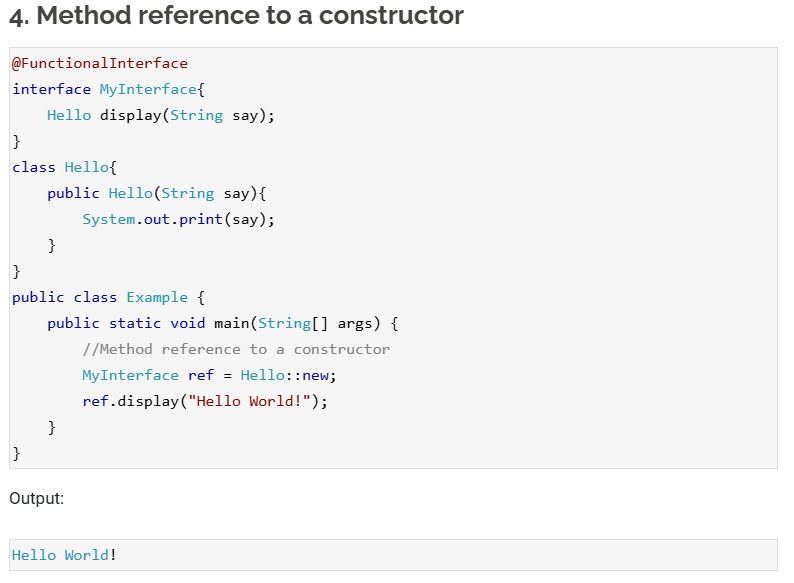
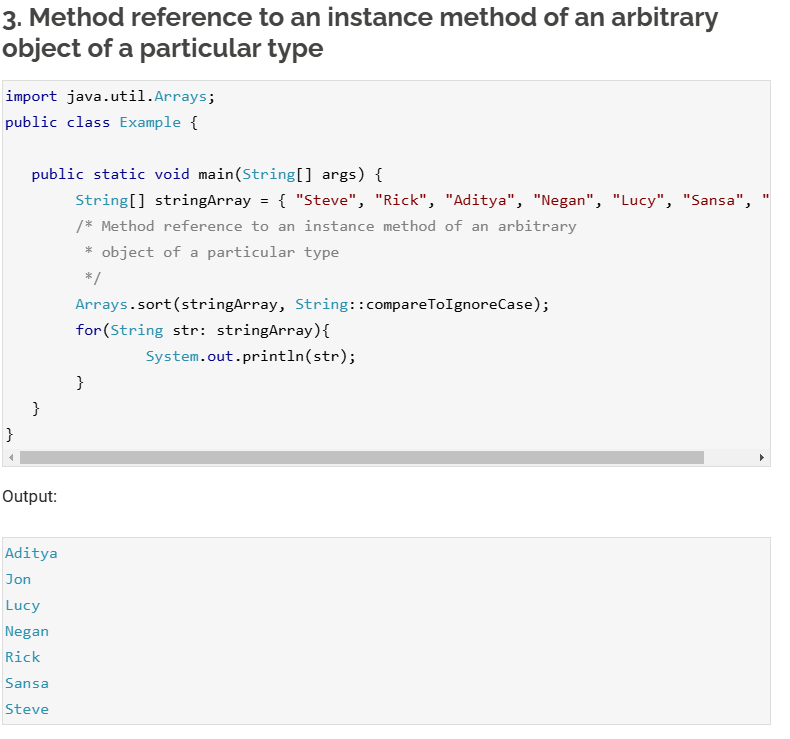
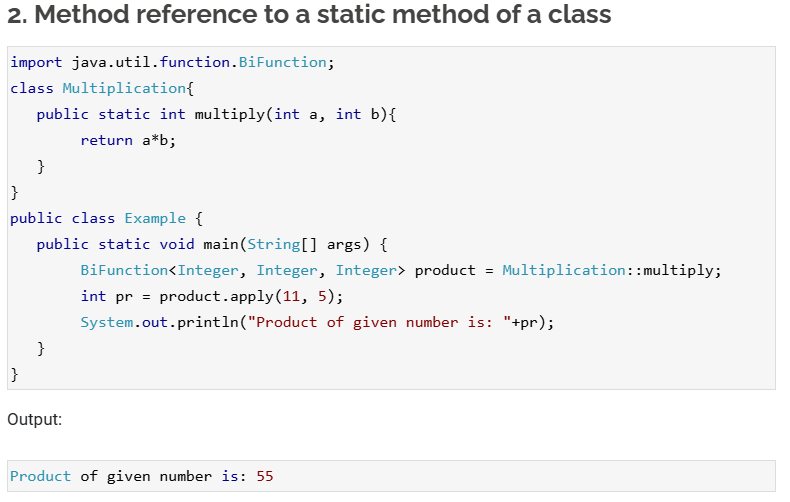
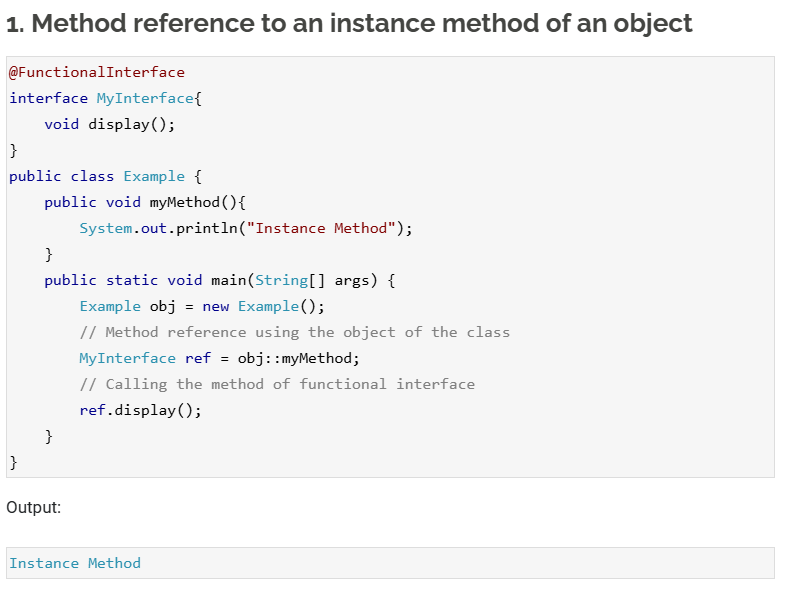
then you can replace it with a method reference like this:

System.out::println

The :: operator is used in method reference to separate the class or object from the method name(we will learn this with the help of examples).

## **Four types of method references**

1. Method reference to an instance method of an object – object::instanceMethod  
2. Method reference to a static method of a class – Class::staticMethod  
3. Method reference to an instance method of an arbitrary object of a particular type – Class::instanceMethod  
4. Method reference to a constructor – Class::new

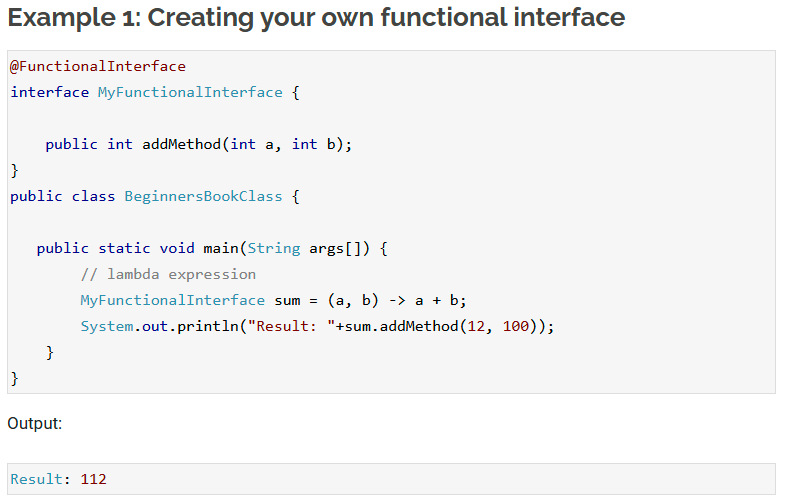


# **Java Functional Interfaces**

* An interface with **only single abstract method** is called functional interface. You can either use the predefined functional interface provided by Java or create your own functional interface and use it.
* they are also known as Single Abstract Method interfaces (SAM Interfaces).
* While creating your own functional interface, mark it with **@FunctionalInterface** annotation, this annotation is introduced in Java 8. Although its optional, you should use it so that you get a compilation error if the interface you marked with this annotation is not following the rules of functional interfaces.

## **What are the rules of defining a functional interface?**

The functional interface should have **Only one** abstract method. Along with the one abstract method, they can have any number of default and static methods.



# **Java 8 Interface Changes – default method and static method**

* Prior to java 8, [**interface in java**](https://beginnersbook.com/2013/05/java-interface/) can only have abstract methods. All the methods of interfaces are public & abstract by default. Java 8 allows the interfaces to have default and static methods. The reason we have default methods in interfaces is to allow the developers to add new methods to the interfaces without affecting the classes that implements these interfaces.

## **Why default method?**

For example, if several classes such as A, B, C and D implements an interface XYZInterface then if we add a new method to the XYZInterface, we have to change the code in all the classes(A, B, C and D) that implements this interface. In this example we have only four classes that implements the interface which we want to change but imagine if there are hundreds of classes implementing an interface then it would be almost impossible to change the code in all those classes. This is why in java 8, we have a new concept “default methods”. These methods can be added to any existing interface and we do not need to implement these methods in the implementation classes mandatorily, thus we can add these default methods to existing interfaces without breaking the code.

We can say that concept of default method is introduced in java 8 to add the new methods in the existing interfaces in such a way so that they are backward compatible. Backward compatibility is adding new features without breaking the old code.

**Static methods** in interfaces are similar to the default methods except that we cannot override these methods in the classes that implements these interfaces.

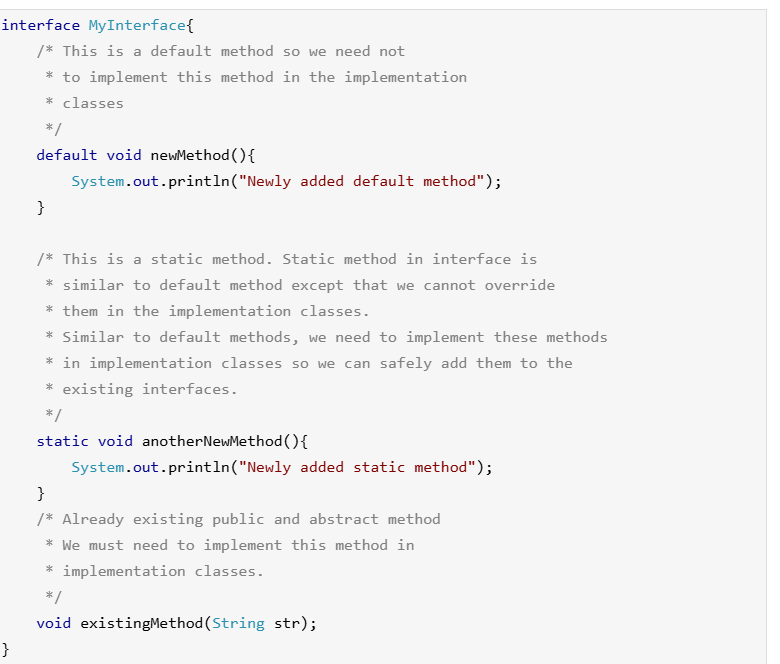
## **Java 8 Example: Default method in Interface**

The method newMethod() in MyInterface is a default method, which means we need not to implement this method in the implementation class Example. This way we can add the default methods to existing interfaces without bothering about the classes that implements these interfaces.



## **Java 8 Example: Static method in Interface**

As mentioned above, the static methods in interface are similar to default method so we need not to implement them in the implementation classes. We can safely add them to the existing interfaces without changing the code in the implementation classes. Since these methods are static, we cannot override them in the implementation classes



## **Java 8 – Abstract classes vs interfaces**

With the introduction of default methods in interfaces, it seems that the [**abstract classes**](https://beginnersbook.com/2013/05/java-abstract-class-method/) are same as interface in java 8. However this is not entirely true, even though we can now have concrete methods(methods with body) in interfaces just like abstract class, this doesn’t mean that they are same. There are still few differences between them, one of them is that abstract class can have constructor while in interfaces we can’t have constructors.

The purpose of interface is to provide full abstraction, while the purpose of abstract class is to provide partial abstraction. This still holds true. The interface is like a blueprint for your class, with the introduction of default methods you can simply say that we can add additional features in the interfaces without affecting the end user classes.

## **Default Method and Multiple Inheritance**

The [**multiple inheritance**](https://beginnersbook.com/2013/05/java-multiple-inheritance/) problem can occur, when we have two interfaces with the default methods of same signature. Lets take an example.



# **Java 8 Stream**

* By using streams we can perform various aggregate operations on the data returned from collections, arrays, Input/Output operations.



## **How to work with Stream in Java**

As we have seen in the above example, the working of stream can be explained in three stages:  
1. Create a stream

2. Perform **intermediate operations** on the initial stream to transform it into another stream and so on on further intermediate operations. In the above example, the filter() operation is intermediate operation, there can be more than one intermediate operations.

3. Perform **terminal operation** on the final stream to get the result. In the above example, the count() operation is terminal operation.

## **Java Stream Features**

1. Stream **does not store** the elements. it simply performs the aggregate operations(such as filter() and count() that we have seen in the above example) to get the desired stream of data.

2. The aggregate operations that we perform on the collection, array or any other data source **do not change** the data of the source, they simply return a new stream. For example the code we have seen above is filtering the strings with length less than 6 using the stream operations but it didn’t change the elements of the list.

3. All the stream operations are **lazy** in nature which means they are not executed until they are needed. For example, if we want to display only the first 2 elements of a list using stream, the stream operation would stop at the end of second iteration after displaying the second element of list.

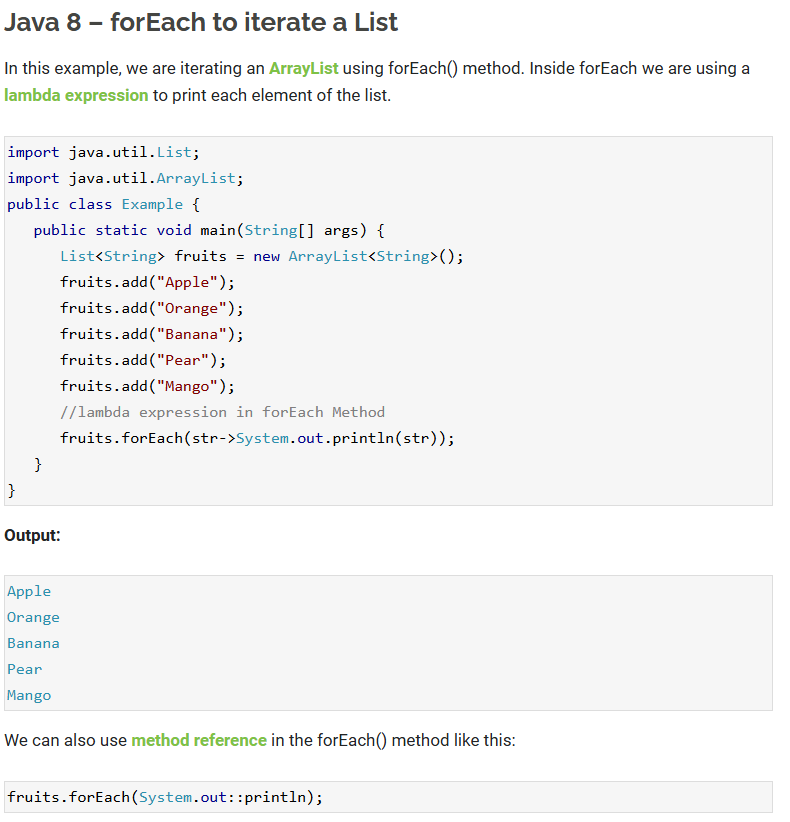
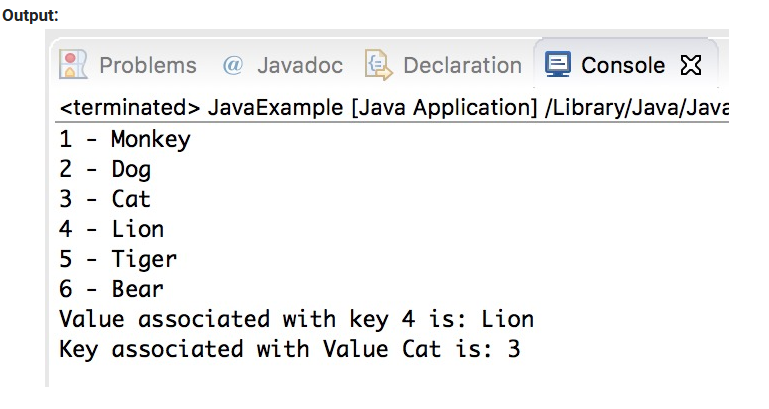


# **Java 8 Stream Filter with examples**

The filter() is an intermediate operation that reads the data from a stream and returns a new stream after transforming the data based on the given condition.



# **Java 8 forEach method with example**



## **Java – Stream forEachOrdered() Method Example**

For sequential streams the order of elements is same as the order in the source, so the output would be same whether you use forEach or forEachOrdered. However when working with parallel streams, you would always want to use the forEachOrdered() method when the order matters to you, as this method guarantees that the order of elements would be same as the source. Lets take an example to understand the difference between forEach() and forEachOrdered().



# **Java 8 – Stream Collectors Class**

* Collectors is a [**final class**](https://beginnersbook.com/2014/07/final-keyword-java-final-variable-method-class/) that extends the Object class.



## **Java Collectors Example – Collecting Data as Set**

In this example we are converting the list of students to the [**stream**](https://beginnersbook.com/2017/10/java-8-stream-tutorial/) and then we are applying the [**Java Stream filter**](https://beginnersbook.com/2017/10/java-8-stream-filter/) to get the selected records from the stream, after that we are converting that stream to set using Collectors.toSet() method.



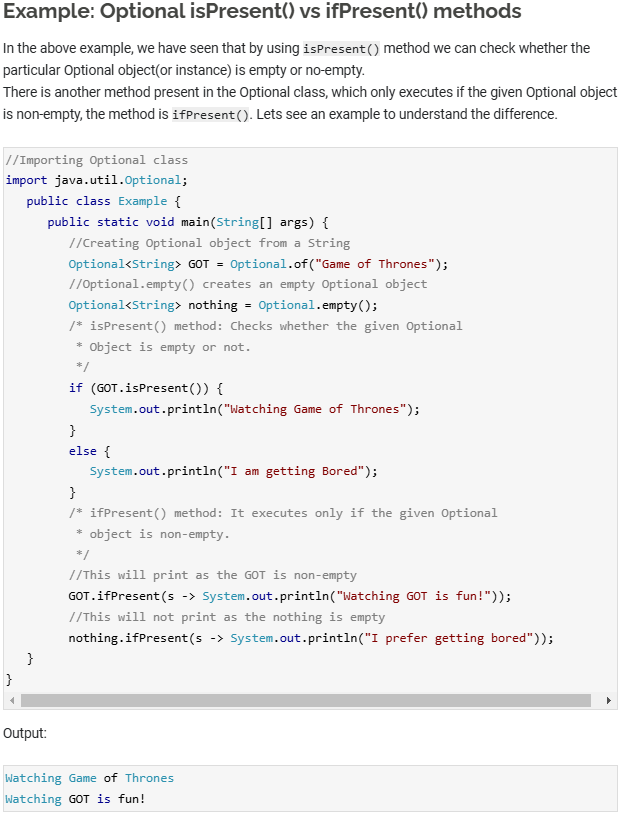
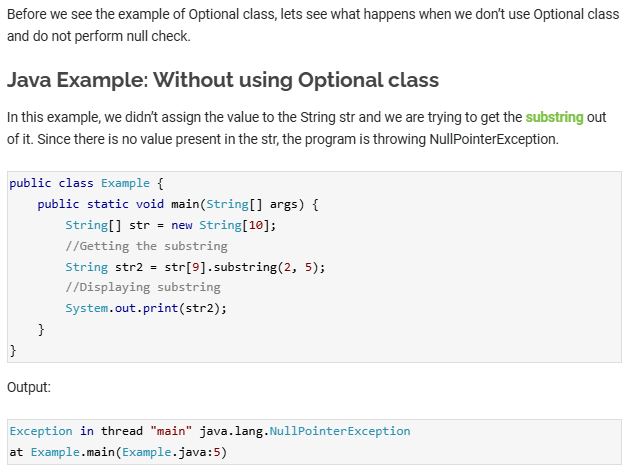
# **Java 8 StringJoiner**

* Using this class we can join more than one strings with the specified delimiter, we can also provide prefix and suffix to the final string while joining multiple strings.



# **Java 8 Optional Class**

* In Java 8, we have a newly introduced Optional class in java.util package. This class is introduced to avoid NullPointerException that we frequently encounters if we do not perform null checks in our code. Using this class we can easily check whether a variable has null value or not and by doing this we can avoid the NullPointerException.



# **Java 8 – Arrays Parallel Sort**

* Java 8 introduced a new method parallelSort() in the Arrays class of java.util package. This method is introduced to support the parallel sorting of array elements.  
  Algorithm of parallel sorting:  
  1. The given array is divided into the sub arrays and the sub arrays are further divided into the their sub arrays, this happens until the sub array reaches a minimum granularity.  
  2. The sub [**arrays are sorted**](https://beginnersbook.com/2024/06/sorting-2d-array-in-java/) individually by multiple threads. The parallel sort uses [**Fork/Join Framework**](https://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html) for sorting sub arrays parallelly.  
  3. The sorted sub arrays are merged.
* The parallelSort() method uses the concept of [**multithreading**](https://beginnersbook.com/2013/03/multithreading-in-java/) which makes it much faster compared to the [**normal sort**](https://beginnersbook.com/2014/07/how-to-sort-an-array-in-java/) when there are lot of elements.

